



12

Research and Development Technical Report
ECOM-76-0329-F

ADA034086

**INTEGRATED SOFTWARE DEVELOPMENT SYSTEM/
HIGHER ORDER SOFTWARE CONCEPTUAL DESCRIPTION**

(Version 1)

M. Hamilton
S. Zeldin

Prime Contractor:

THE CHARLES STARK DRAPER
LABORATORY, INC.
555 Technology Square
Cambridge, MA 02139

Subcontractor:

HIGHER ORDER SOFTWARE, INC.
843 Massachusetts Avenue
Cambridge, MA 02139

NOVEMBER 1976

Final Report for Period November 1975 - November 1976

DISTRIBUTION STATEMENT

Approved for public release;
distribution unlimited.

PREPARED FOR:

ECOM



CENTACS

ARMY ELECTRONICS COMMAND, FORT MONMOUTH, NEW JERSEY 07703

DDC
RECEIVED
JAN 8 1977

NOTICES

Copyright © 1976 by —

HIGHER ORDER SOFTWARE, INC.

All rights reserved.

No part of this report may be reproduced in any form, except by the U.S. Government, without written permission from Higher Order Software, Inc. Reproduction and sale by the National Technical Information Service is specifically permitted.

DISCLAIMERS

The findings in this report are not to be construed as an official Department of the Army position, unless so designated by other authorized documents.

The citation of trade names and names of manufacturers in this report is not to be construed as official Government indorsement or approval of commercial produces or services referenced herein.

DISPOSITION

Destroy this report when it is no longer needed. Do not return it to the originator.

INTEGRATED SOFTWARE DEVELOPMENT SYSTEM/
HIGHER ORDER SOFTWARE CONCEPTUAL DESCRIPTION
(Version 1)
M. Hamilton and S. Zeldin

ERRATA SHEET

Modifications to the above report are as follows:

Page ix: Insert after 4.3.6: Multilevel Set Partition

4.3.7: An Example of Class Partition

Page 75: Insert after Figure 4.3.6: Multilevel Set Partition:

Class partition is illustrated in Figure 4.3.7. While set partition involves partition of the domain into subsets, class partition involves partition of the domain variables into classes and the partition of the range variables into classes. In the example, it is assumed that the domain variable has an associated data structure comprised of two parts, x_1 and x_2 . Likewise, the range variable has an associated data structure with the same number of classes as the domain's data structure. (As an example of such a structure, consider the domain to be the complex numbers; the range to be polar coordinates. Then, for a given value of the domain variable (i.e., a given complex number), x_1 would represent its real part and x_2 its imaginary part.) Consequently, the variable is partitioned into two separate classes, x_1 and x_2 , such that elements associated with x_1 are the input elements that one offspring can access and the elements associated with x_2 are the input elements that the other offspring can access. The range structure is partitioned in a similar manner.

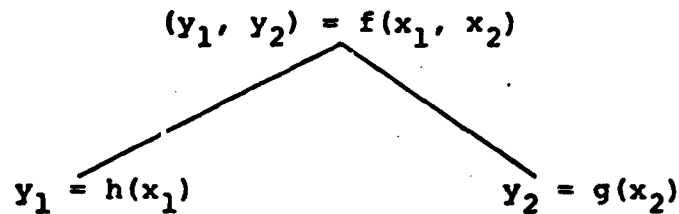


Figure 4.3.7: An Example of Class Partition

The following characteristics with respect to class partition should be observed:

- (1) All offspring of the module at f are granted permission to receive input values taken from a partitioned variable in the set of the parent MCF domain variables, such that each offspring's set of input variables are non-overlapping and all the offspring input variables collectively represent only its parent's MCF input variables.
- (2) All offspring of the module at f are granted permission to produce output values for a partitioned variable in the set of the parent MCF range variables, such that each offspring's set of output variables are non-overlapping and all the offspring's output variables collectively represent the parent MCF output variables.
- (3) Each offspring is specified to be invoked such that for each change in state of its parent, all offspring undergo a state change.
- (4) There is no communication between offspring.

19 REPORT DOCUMENTATION PAGE		READ INSTRUCTIONS BEFORE COMPLETING FORM
1. REPORT NUMBER ECOM 76-0329-F	2. GOVT ACCESSION NO.	3. RECIPIENT'S CATALOG NUMBER
4. TITLE (and Subtitle) Integrated Software Development System/ Higher Order Software Conceptual Description (ISDS/HOS)		5. TYPE OF REPORT & PERIOD COVERED Final Report, For Period Nov 75 - Nov 76
6. PERFORMING ORG. REPORT NUMBER Higher Order Software Technical Report # 3		7. CONTRACT OR GRANT NUMBER(s) DAAB07-76-C-0329
8. AUTHOR(s) M. Hamilton S. Zeldin		9. PERFORMING ORGANIZATION NAME AND ADDRESS Prime Contractor: The Charles Stark Draper Laboratory 555 Technology Square Cambridge, MA 02139 Subcontractor: Higher Order Software, Inc. 843 Massachusetts Avenue Cambridge, MA 02139
10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS 157.62701.AH92.B1.33		11. CONTROLLING OFFICE NAME AND ADDRESS Software Engineering Div. (DRSEL-NL-BG) Center for Tactical Computer Sciences (CENTACS) ECOM, Ft. Monmouth, N. J. 07703
12. REPORT DATE November 1976		13. NUMBER OF PAGES 335 332 p.
14. MONITORING AGENCY NAME & ADDRESS (if different from Controlling Office) Software Engineering Div. (DRSEL-NL-BG) Center for Tactical Computer Sciences (CENTACS) ECOM, Ft. Monmouth, N. J. 07703		15. SECURITY CLASS. (of this report) UNCLASSIFIED 15a. DECLASSIFICATION/DOWNGRADING SCHEDULE
16. DISTRIBUTION STATEMENT (of this Report) Approved for public release, distribution unlimited.		
17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report) 15762701AH92 EL		
18. SUPPLEMENTARY NOTES		
19. KEY WORDS (Continue on reverse side if necessary and identify by block number) formal systems, system development process, reliability, axioms, specification, control, methodology, decomposition, automated tools, systems development disciplines		
20. ABSTRACT (Continue on reverse side if necessary and identify by block number) The Integrated Software Development System/Higher Order Software (ISDS/HOS) is a unified systems engineering methodology that includes a basic set of principles and a standard set of tools and techniques for developing computer-based systems. The foundation of ISDS/HOS is based upon a formal system theory, Higher Order Software (HOS). The principles of ISDS/HOS encompass all phases of system de- velopment and all disciplines including design, verification, documentation, management, and maintenance.		

The automated tools of ISDS/HOS serve to eliminate many possible sources of human error in the transition of system development from concept to deployment. The component tools of ISDS/HOS facilitate the specification of a system, automate interface analysis of the specification, and automate translation of the system specification directly into an optimized target-machine coded form. The ISDS/HOS support tools eliminate much of the manual effort required in the development of computer-based systems.

The purpose of this report is to present the ISDS/HOS concept so that managers, systems engineers, and computer scientists can appreciate the various aspects of the basic principles on which ISDS/HOS is based, the basic tools that ISDS/HOS will have available, and the basic techniques that ISDS/HOS users can use.

ADDITIONAL

WTR	WTR	<input checked="" type="checkbox"/>
DOC	DOC	<input type="checkbox"/>
UNCLASSIFIED		<input type="checkbox"/>
JUSTIFIED		<input type="checkbox"/>
B. DISSEMINATION/AVAILABILITY		
A		

ACKNOWLEDGEMENTS

This report was prepared under Contract No. DAAB07-76-C-0329 sponsored by the Department of the Army, Headquarters United States Army Electronics Command, Fort Monmouth, New Jersey and subcontracted to Higher Order Software, Inc. by The Charles Stark Draper Laboratory, Inc.

We would like to extend a special acknowledgement to Marty Wolfe and Ed Lieblein of CENTACS, ECOM/Ft. Monmouth, New Jersey for both providing major sections to Chapter 4 as well as providing a very constructive and critical review of the entire report.

With respect to the staff of Higher Order Software, Inc., we would like to thank, in particular, Chris Davis and Bill Heath for their technical contributions in Chapter 6; Chris Davis for his technical contributions in Chapter 2; and Bill Heath for Chapter 7. In addition, we would like to thank Roy Coppinger for his help in preparing written material for Chapters 1, 3, and 6; Cosmo Battinelli for his help in preparing written material for Chapters 2 and 6; and Steve Cushing for his comments which were very helpful in the preparation of this report.

We would also like to thank G. Bender of Hughes Aircraft Corp. for Section 2.3.1; N. Hampton and G. Myers of Naval Electronics Laboratory Center for Section 2.3.2; D. Teichroew and E. Hershey III, of the University of Michigan, for Section 2.3.2; T. Straeter of NASA/Langley Research Center for Section 2.3.7; and E. Damon of NASA/Goddard Spacecraft Center for Section 2.3.8.

In addition, we would also like to thank D. DeVorkin of The Charles Stark Draper Laboratory for contributing material in

iii

Sections 2.0, 2.1, and 2.2 and also R. Freiburghous of Translation Systems, Inc. for contributing the major portions of Section 6.3.3.

The authors would like to express appreciation to Gail Lopes, Andrea Davis, Virginia Wier, and Adele Volta for the preparation of this report; to Lauren Hamilton for the artwork of Chapter 4; and to Geoffrey Gold and Gail Lopes for the artwork of Chapter 5.

Our preliminary work was performed at The Charles Stark Draper Laboratory prior to the incorporation of Higher Order Software, Inc.

Margaret Hamilton and Saydean Zeldin

TABLE OF CONTENTS

	<u>Page</u>
1.0 INTRODUCTION	1
2.0 STATEMENT OF THE PROBLEM	5
2.1 Current DoD Efforts	11
2.2 Commercially Available Software Development Aids	15
2.3 Current System and Software Development Techniques and Methodologies	19
2.3.1 Structured Design	19
2.3.2 System Design Laboratory	22
2.3.3 Information System Design and Optimization System (ISDOS)	23
2.3.4 Software Factory	29
2.3.5 Ballistic Missile Defense (BMD) Software Development System (SDS)	35
2.3.6 Hierarchy plus Input-Process-Output (HIPO)	42
2.3.7 Multipurpose User-Oriented Software Technology (MUST)	48
2.3.8 Domonic	52
2.3.9 Summary	53
3.0 RATIONALE FOR ISDS/HOS	55
3.1 Background	57
3.2 Concept of the HOS Formalized Approach	59
4.0 FOUNDATIONS OF ISDS/HOS	61
4.1 Preliminaries	63
4.1.1 Trees and Functions	63
4.1.2 Modules and Nodal Families	67
4.2 The Axioms	68
4.3 Functional Decomposition	72
4.4 Illustration of the Axioms	75
4.5 Examples	85
4.5.1 The BRIGGEN System	85

	<u>Page</u>
4.5.2 The Line Justifier	100
5.0 THE USE OF ISDS/HOS DURING THE LIFE-CYCLE OF COMPUTER-BASED MILITARY SYSTEMS	111
5.1 Systems Preliminaries	117
5.2 ISDS/HOS Disciplines for Use in Developing a System	127
5.3 ISDS/HOS Development Phases for a System	137
5.3.1 Concept Formulation Phase	145
5.3.2 Program Validation Phase	152
5.3.3 Full Scale Development Phase	158
5.3.4 Production and Deployment Phase	168
5.4 Tools Used During the Phases of System Development	169
5.5 System Building Process	172
5.5.1 ACS Demonstrated by System Layer Function S	176
5.5.2 The Management Building Layer (M)	181
5.5.3 Subsystems of System S	184
5.5.4 The Environment Layer Subsystem	186
5.5.5 The Application System Layer S _A	191
5.5.6 The Building Levels of Support Tools Functions	194
5.5.7 'Frozen' Modules	194
6.0 TOOLS FOR ISDS/HOS	197
6.1 Component Tools of ISDS/HOS	199
6.1.1 Specification Language (AXES)	199
6.1.2 Design Analyzer	207
6.1.3 Static Resource Allocation Tool (RAT)	215
6.1.4 Structuring Executive	228
6.2 Support Tools	233
6.2.1 Management Support Tools	233
6.2.2 Documentation Support Tools	248

	<u>Page</u>
6.2.3 Design Support Tools	250
6.3 Incremental Tools for Current Use of ISDS/HOS	259
6.3.1 Assembly Language	259
6.3.2 Macro Processors/Assemblers	261
6.3.3 Higher Order Language (HOL)	262
6.3.4 Compilers	266
6.3.5 Structured Design Diagrammer	268
6.3.6 Interactive Debugger	274
6.3.7 Interpreter	274
7.0 CONCLUSIONS	277

BIBLIOGRAPHY

APPENDIX I. Definitions and Properties of Control

APPENDIX II. Derivation of the Primitive Control Structures

FIGURES

	<u>Page</u>
2.1 Interrelation of Software Acquisition Study Findings	9
2.2 Problem Summary	10
2.3.6.1 The HIPO Technique	46
4.1.1.1 An Example of a Tree Structure	63
4.1.1.2 Tree Levels	64
4.1.1.3 Parent-Offspring Relationship	64
4.1.1.4 Tree Substructures	65
4.1.1.5 Illustration of a Function from X into Y	67
4.3.1 An Example of Composition	72
4.3.2 Composition with Three Functions on One Level	73
4.3.3 Multilevel Composition	73
4.3.4 An Example of Set Partition	74
4.3.5 Set Partition with Three Functions on One Level	75
4.3.6 Multilevel Set Partition	75
4.4.1 Axiom 1 - Invocation Rights	77
4.4.2 Axiom 2 - Responsibility Rights	79
4.4.3 Axiom 3 - Output Access Rights	80
4.4.4 Axiom 4 - Input Access Rights	81
4.4.5 Axiom 5 - Rejection Rights	82
4.4.6 Axiom 6 - Ordering Rights	84
4.5.1 BRIGGEN Invocation Tree	86
4.5.2 BRIGGEN Management Structure Control Map	88
4.5.3 BRIGGEN Time Requirements	95
4.5.2.1 The Initial Assumption of Line Justifier together with its Supporting Narrative	101
4.5.2.2 Line Justifier Decomposed Using the Composition Primitive Control Structure	102

	<u>Page</u>
4.5.2.3 F_1 Decomposed Using the Set Partition Primitive Control Structure	103
4.5.2.4 Set Partition Decomposition of F_2 to Return Lines Containing Only One Word	104
4.5.2.5 Decomposition Using the Composition Primitive Control Structure	105
4.5.2.6 Restructuring of the Control Map to Include Line Parity Requirement	106
4.5.2.7 Line Parity Set Partition of F_4	108
4.5.2.8 Invocation Map for Line Justifier	109
4.5.2.9 HOL Code for Line-Justifier System	110
5.1.1 System A	117
5.1.2 System TRANSLATE	118
5.1.3 An Example of a Support System Function COMPILER which Creates a New Layer for System A	120
5.1.4 The OS System Translates (a) and Returns Execution Control Back to System, A (b)	121
5.1.5 Dynamic Translation Process	122
5.2.1 Development	127
5.2.2a Development Process - One Phase	130
5.2.2b Design_Implementation Disciplines	131
5.2.2c Verification Discipline	132
5.2.2d Management Discipline	133
5.2.3 Example of Management Discipline Between Steps of One Phase Within an Integrated System Development Process	135
5.3.1 Four Major Phases of a System Development	138
5.3.2 Delivery of Requirements for Next Phase	138
5.3.3 Example of System Management Techniques	140
5.3.4 The Top-Level Design and Implementation Disciplines of an Integrated System Development Phase	143
5.3.5 The Top-Level Verification Disciplines of an Integrated System Development Process	144

	<u>Page</u>
5.3.1.1 One Step of Concept Formulation Phase	147
5.3.1.2 Examples of Iterative Steps Within the Concept Formulation Phase	149
5.3.1.3 Concept Formulation Phase Specification Process	150
5.3.2.1 Target System AXES _S	153
5.3.2.2 An Example of Top Layer Resource Allocation for Target System AXES _S	154
5.3.2.3 One Step of the Program Validation Phase	156
5.3.3.1 Potential Iterative Steps Within the FSD Phase	159
5.3.3.2 An Example of One Step of the Full Scale Development Phase (For the Incremental Model)	160
5.3.3.3 Major Translation Steps of an Integrated System Development Process	163
5.3.3.4 Translation Process for System A	165
5.3.3.5 Development and Execution Layers of System A	167
5.3.4.1 An Example of Processes of the Production and Deployment Phase	170
5.5.1 Building Matrices for Various Levels of System Development	175
5.5.1.1 Building Process for System S with Respect to Personnel Management and the Translation Process	177
5.5.1.2 The Assembly Control Supervisor Concept	179
5.5.1.3 Building the Library and the Use of the Library Managed by ACS	180
5.5.2.1 System Layer Function M With Respect to System S and the Personnel Management	183
5.5.4.1 Sample Building Process for Layer System Function E	188
5.5.5.1 Building the Applications System	193

	<u>Page</u>
5.5.7.1 ISDS/HOS Building Process	196
6.1.2.1 Top Level Decomposition of Analyzer Tool	208
6.1.2.2 Assertions for Composition Primitive	211
6.1.3.1 The Primitive Control Structure: Composition	217
6.1.3.2 The Primitive Control Structure: Set Partition	219
6.1.3.3 The Primitive Control Structure: Class Partition	220
6.1.3.4 Condensing of the Composition Primitive	221
6.1.3.5 Condensing of the Class Partition Primitive	222
6.1.3.6 Time-Optimal Resource Allocation	225
6.1.4.1 Dynamic System Reconfiguration	229
<u>6.2.1.1.1</u> Structure of ISDS/HOS Project System Data Base	236
<u>6.2.1.2.1</u> Simplified ISDS/HOS Decomposition Map	239
<u>6.2.1.3.1</u> Inter-Revision Updater Example	243
<u>6.2.1.4.1</u> User Creation of New System via the ISDS/HOS Collector	247
<u>6.2.3.1.1</u> Top-Level ISDS/HOS Decomposition of a Simulator	251
<u>6.2.3.3.1</u> Summary of Procedure for Producing Optimal Specification for Emulator	258
6.3.5.1 A Structured Program Using Standard Symbols to Show Flow of Program Execution	269
6.3.5.2 Same Structured Program Using Structured Flowchart Conventions	270
6.3.5.3 Design Diagram Notation	273

TABLES

	<u>Page</u>
2.2.1 Commercially Available Software- Development Support Tools	16
4.2.1 Axioms of ISDS/HOS	71
5.1.1 Requirements Affecting the Development of a Target System	124
5.1.2 Requirements of the Target System	126
5.3.1.1 Concept Formulation Phase Tools and Techniques	148
5.3.1.2 Concept Formulation Phase Specification Process Tools and Techniques	151
5.3.2.1 Program Validation Phase Tools and Techniques	157
5.3.3.1 Tools and Techniques Applied Within One Step of the FSD Phase	161
5.4.1 Tools Used During the Phases of a System Development	171
5.5.1 System Building Matrix Used by Transla- tion Project Management Personnel to Track Translation Development	173
5.5.3.1 System Building Matrix for Subsystem of S	185
5.5.4.1 Building Matrix for System S _E , The Environment System Layers	187
5.5.5.1 Development Layers	192
5.5.6.1 Building Matrix for ISDS/HOS Support Tools	195
6.1.1.1 Characteristics of Proper Specifications	200
6.1.3.1 Input and Output of the Resource Allocation Tool	218
<u>6.2.1.3.1</u> Files Maintained by Inter-Revision Updater	244
<u>6.2.1.3.2</u> Statistics Automatic- ally Collected by the Inter- Revision Updater	245
<u>6.2.3.1.1</u> Characteristics of an ISDS/HOS Simulator	253

1.0 INTRODUCTION

1.0 INTRODUCTION

The rapidly increasing cost of software represents a serious threat to the effectiveness of future systems for the Department of Defense. Already software cost is a major component of overall systems-development cost, and this trend is expected to accelerate in the near future. As systems grow in complexity, there is an increasing dependence on sophisticated computer software to support them. It is widely recognized that present methods of software development are not sufficient to produce reliable software systems at an acceptable cost (RAM75) (WU74) (BOE72) (GAN76). Reliable software is particularly important for tactical systems required to respond to multiple threats in a real-time environment. A significant improvement in software-development technology is required to ensure the success of future large-scale systems. In order to satisfy this requirement, the Army has identified the need for an Integrated Software Development System/Higher Order Software (ISDS/HOS) for the development of reliable, maintainable, versatile computer systems at a significantly reduced life-cycle cost.

This document presents ISDS/HOS as a comprehensive approach to the solution of the software-development problem. The problem is defined in Chapter 2 by presenting available methodologies and techniques for system and software development. A rationale of ISDS/HOS is presented in Chapter 3, by means of its historical development. Chapter 4 then describes the theory of Higher Order Software which is the foundation of ISDS/HOS. Chapter 5 describes the use of ISDS/HOS during the life-cycle of computer-based systems. This includes the step-by-step transformation of user requirements into hardware, software, and firmware designs. Chapter 6 describes the automated tools of ISDS/HOS.

2.0 STATEMENT OF THE PROBLEM

2.0 STATEMENT OF THE PROBLEM

Concern with software costs is evidenced in a DoD memorandum (DDR74) which presents the trend in the increasing ratio of software to hardware costs. The memorandum further references a RAND forecast which predicts a 95% expenditure of the ADP budget on software by 1985. Without being concerned over the accuracy of the forecast, it is apparent that the trend of increasing software cost is expected to continue. A later DoD memorandum (DDR75) states that "There is an urgent need for technical and managerial innovations which lead to more reliable and cost-effective software throughout DoD." The memorandum then places emphasis on the areas of command and control and weapons-system software.

The increasing ratio of software to hardware costs can be attributed to (1) historical emphasis placed on hardware technology and its attendant design and development methodologies, and (2) the lack of similar rigor placed on the software technology and its attendant design and development methodologies. Of particular interest is the fact that the hardware required for a system was generally designed and, sometimes, developed prior to even considering the required software. This development process forced the software not only to perform its required functions, but, also, to make up any deficiencies in the hardware and/or changes in requirements. As a result, the software became complex, specific to the mission, inflexible, costly, and untimely. A review of previous system development efforts indicates:

- software development costs are much higher than expected, and almost always exceed initial estimates.
- schedules are undependable, resulting in excessive system-development cycle time.
- reliability of the systems developed is unsatisfactory.
- computer resource requirements almost always seem to exceed initial estimates.
- measures of correctness are inadequate.

- communication between users is poor due to lack of visibility.
- the resultant software is often inflexible, which impedes correction, transferability, and enhancement.
- the resultant system is frequently less useful than initially desired.
- developers and users are uncertain as to the amount of testing required; frequently redundant (unnecessary) tests are conducted which are costly, or tests are omitted (which eventually, are probably more costly).
- management problems are all too frequent in large systems.
- documentation in large systems becomes unwieldy.
- requirements never seem to get nailed down (made precise and stabilized).

Figures 2.1 (KOS75) and 2.2 (DER75) summarize the findings of a joint study on "DoD Weapons Systems Software Acquisition and Management" performed by MITRE (ASC75) and the Applied Physics Laboratory at Johns Hopkins University (KOS75). Various combinations of the problems indicated in these figures are plaguing the DoD in current systems-design and development activities.

The remaining sections of this chapter present various methods (including approaches and guidelines) used by the DoD and commercial industries, and methodologies available to the DoD which may alleviate certain of the problems mentioned above.

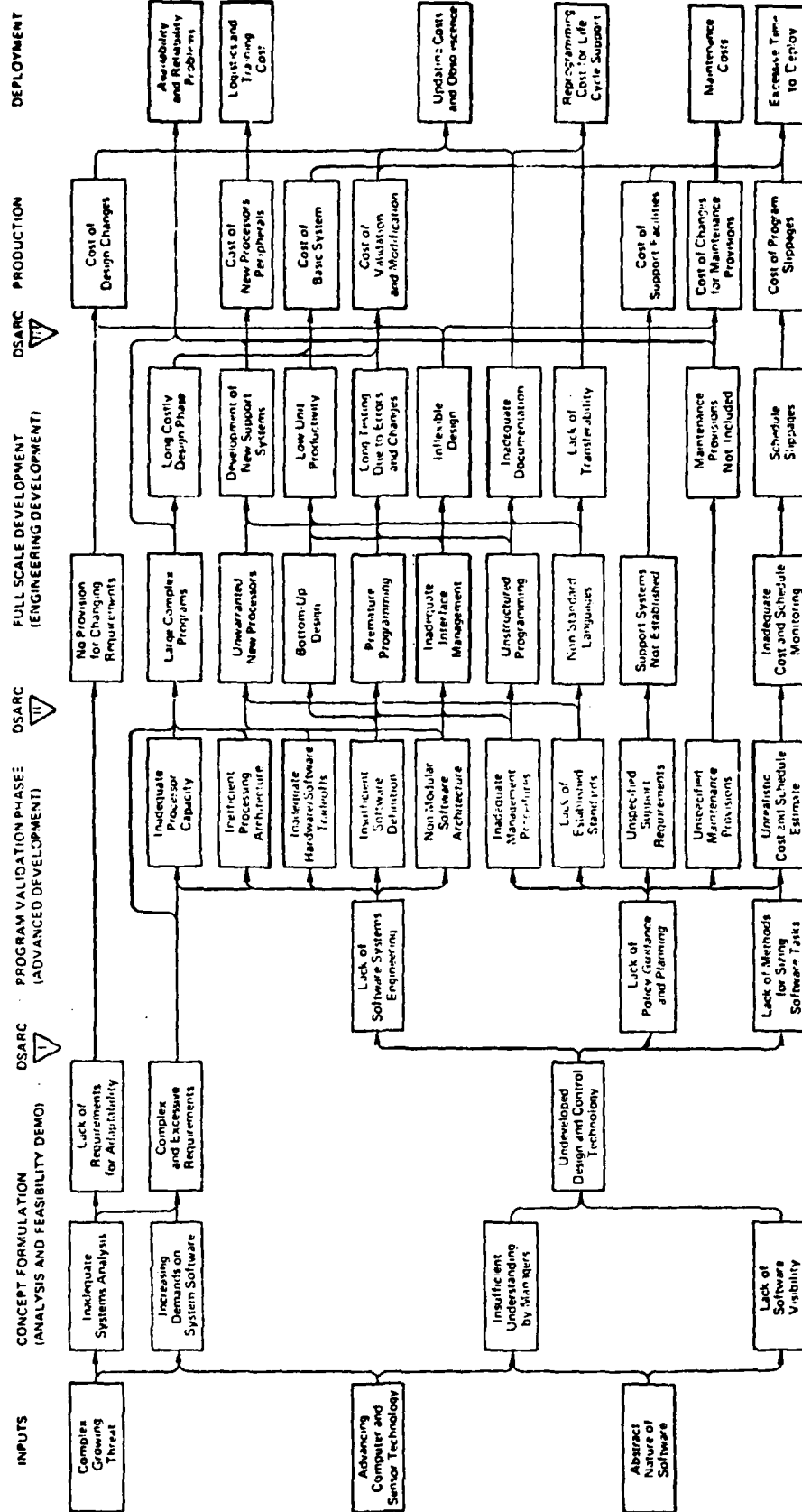


Fig. 2.1 Interrelation of Software Acquisition Study Findings

<p>Visibility in weapon system acquisition</p> <ul style="list-style-type: none"> ● Inadequate requirements analysis ● Inadequate interface management ● Inadequate documentation ● Lack of transferability ● Inaccurate cost/schedule projections ● Low quality 	<p>Quality assurance and control</p> <ul style="list-style-type: none"> ● Lack of management monitoring of software reliability ● Lack of software reliability quality assurance discipline ● Lack of quantitative data base
<p>Language selection</p> <ul style="list-style-type: none"> ● Low correlation of machine-oriented language to engineering problem ● Lack of design visibility ● Machine dependence 	<p>Lack of software acquisition management standards</p> <ul style="list-style-type: none"> ● Terminology ● Directive, instructions, standards
<p>Language proliferation</p> <ul style="list-style-type: none"> ● Difficult learning process ● Discourages development of test and support tools ● Reduces management visibility ● Complicates institutional control ● Cost redundancy 	<p>Lack of acquisition, management, operations and support guidelines</p> <p>Lack of formal personnel development and training</p> <p>Research and development</p> <ul style="list-style-type: none"> ● Lack of focus ● Relevancy ● Lack of technology base ● Redundancy and duplication

Figure 2.2: Problem Summary

2.1 Current DoD Efforts

DoD is cognizant of the problems encountered in developing software for weapon systems. The DoD Software Management Steering Committee in July 1975 issued a Statement of Principles known as the Capstone Directive (which later evolved into DoD Directive 5000.29). The document seeks to make the role of software explicit, visible, and controlled in the weapon-system acquisition process.

The Committee also provided for studies (KOS75) (ASC75) to be conducted on the state of software acquisition, development, and management for DoD weapon systems. These studies provide a strong statement of technical and management problem areas and supply recommendations on directions along which solutions may be found. The following quotation is from the MITRE study report:

The major contributing factor to weapon system software problems is the lack of discipline and engineering rigor applied consistently to the software acquisition activities.

(One of the lessons learned is that DoD must provide a level of staff support commensurate with the detail and expertise required to monitor tightly the range of activities related to the software-development process in weapon-system acquisition.)

Studies (ASC75) (KOS75) have shown that early and adequate planning is essential to the proper cost-effectiveness of weapon-system software. Planning is an intangible area which requires a large amount of money for "concept" rather than product. There is always pressure, therefore, to compromise this aspect of weapon-system development.

DoD is aware of the problems in cost, management, and technical development of software in general, and software for weapon systems in particular, as reflected in DoD Directive 5000.29. Due to its importance, the policy stated in this directive is reproduced below:

V. Policy

A. General

1. Annual expenditures by DoD in the design, development, acquisition, management, and operational support of computer resources embedded within, and integral to weapons, communications, command and control, and intelligence sensor systems are measured in the billions of dollars. Unreliability, particularly of software, diminishes DoD mission effectiveness in many major Defense systems.
2. Computer resources in Defense systems must be managed as elements or subsystems of major importance during conceptual, validation, full-scale development, production, deployment, and support phases of the life cycle, with particular emphasis on computer software and its integration with the surrounding hardware.

B. Requirements Validation and Risk Analysis

1. Validation of computer resource requirements, including software, risk analyses, planning, preliminary design, security where applicable (DoD Directive 5200.28, reference (f)) and interface control and integration methodology definition will be conducted during the Concept Formulation and Program Validation phases of Defense system development, prior to Defense Systems Acquisition Review Council (DSARC) II.
2. This analysis must assure conformance of planned computer resources with stated operational requirements.
3. Risk analysis, preliminary design, hardware/software integration methodology, external interface control, security features (DoD Directive 5200.28, reference (f)), and life cycle system planning shall be included in the review.
4. Correctness of software, reliability, integrity, maintainability, ease of modification, and transferability will be major considerations in the initial design.

5. The risk areas, and a plan for their resolution shall be included in the Decision Coordinating Paper (DoD Directive 5000.2, Reference (g)).
 6. In addition, computer resource requirements will be continuously coordinated and reconciled with system operational requirements throughout system development after DSARC II.
- C. Configuration Management of Computer Resources. Defense system computer resources, including both computer hardware and computer software will be specified and treated as configuration items. Baseline implementation guidance for this action is contained in DoD Instruction 5010.21 (reference (i)).
- D. Computer Resource Life Cycle Planning. A computer resource plan will be developed prior to DSARC II, and will be maintained throughout the life cycle. The purpose of the plan is to identify important Defense system computer resources acquisition and life cycle planning factors, both direct and indirect; and to establish specific guidelines to ensure that these factors are adequately considered in the acquisition planning process. Examples of factors to be addressed are the following, as applicable:
1. Responsibilities for integration of computer resources into the total Defense system and the determination of overall system quality and integrity.
 2. Personnel requirements for developing and supporting computer resources.
 3. Computer programs required to support the development, acquisition, and maintenance of computer equipment and other computer programs.
 4. Provisions for the transfer of program management responsibility after initial system operating capability has been achieved; provisions for system/equipment turnover.
- E. Support Software Deliverables. Unique support items required to cost effectively develop and maintain the delivered computer resources over the system's life cycle will be specified as deliverable, with DoD acquiring rights to their design and/or use. Examples of such support items are compilers, environmental simulators, documentation aids, test case generators and analyzers, and training aids. The provisions of ASPR, section IX (reference (j)) shall govern the implementation of the policy.

- F. Milestone Definition and Attainment Criteria. Specific milestones to manage the life cycle development of computer resources, including computer system and support software, will be used to ensure the proper sequence of analysis, design, implementation, integration, test, documentation, operation, maintenance, and modification. These milestones will include specific criteria that measure their attainment.
- G. Software Language Standardization and Control. DoD approved High Order Programming Languages (HOLs), (reference (k)) will be used to develop Defense system software, unless it is demonstrated that none of the approved HOLs are cost effective or technically practical over the system life cycle. Each DoD approved HOL will be assigned to a designated control agent who will be responsible for such activities as validating compliance of compiler implementations with the standard language specifications, gathering data as to the use of the language, and for disseminating information, compilers, and tools. The designated control agent will also be responsible for assuring language stability except for DoD HOL specifications which already fall within the purview of DoD Manual 4120.3M (reference (m)).

In addition to the policy stated above, DoD Directive 5000.29 established the DoD Management Steering Committee for Embedded Computer Resources (MSC-ESR), and included the charter of the MSC-ESR in the directive. One of the objectives of the MSC-ESR, as stated in its charter, is to

Formulate a coordinated DoD Technology Base Program for software basic research, exploratory development, advanced development, and technology demonstrations addressing critical software issues that can be recommended to the Director, Defense Research and Engineering.

The net result of the management policy just stated is to:

- place software on a par with hardware throughout the acquisition process.
- provide guidelines for DoD management and technical staff in developing weapons-system acquisition programs conducive to management and technical review.
- provide qualified DoD staff for management and technical review.

- provide software tools (HOL's and support software) for weapons systems development, thus reducing training, development, and review staff requirements.
- establish a coordinated software research and development program addressing critical software issues.

The environment which will be provided by DoD Directive 5000.29 will greatly facilitate the weapons-system acquisition process. MITRE, APL, CENTACS (LIE73) (CEN75), and others (GAN76) (R&D76), however, recognized that technical information, guidelines, management concepts, and directives alone will not solve the overall software problem. Flexible and effective software-development tools, within the context of a coherent, thorough and rigorous development methodology, have been perceived as the technology base necessary to surmount the problem.

2.2 Commercially Available Software Development Aids

Computer-based systems found in the military have counterparts in the commercial marketplace, including:

- Business applications such as accounting and inventory control.
- Scientific computation for research and development.
- Large data-base applications with the requirement for data-base management and management information-system software.
- Real-time, stimulus-response applications for command and control, communications, signal processing, and process control.

It is important to realize that commercial organizations, including computer manufacturers with their associated software development divisions and software vendors, are able to approach software and systems development with the advantages of continuity of personnel over time and identifiable common requirements for

a particular set of users. For these reasons, it is possible for commercial organizations to develop a certain amount of applications-software support tools to aid their users in software and systems development. Table 2.2.1 lists examples of these software tools.

- HIGHER-ORDER LANGUAGES/COMPILERS
- ASSEMBLERS AND MACRO-ASSEMBLERS
- LINKAGE EDITORS AND RELOCATABLE LOADERS
- LIBRARIES OF COMMON SYSTEM-UTILITY ROUTINES
- LIBRARIES OF COMMON APPLICATIONS ROUTINES
- DATA-BASE MANAGEMENT AND MANAGEMENT-INFORMATION SYSTEMS
- INTERACTIVE MAN-MACHINE INTERFACE SUPPORT

TABLE 2.2.1: Commercially Available Software-Development Support Tools.

Due to the availability of these software-development support tools, many users of commercial computer facilities and organizations have been able to realize cost benefits in their applications and systems-development efforts. These users share only marginally in the costs of tool development, but obtain the total benefit of each tool. It is clear that each of the tools (especially each common routine) used in a development project is simply one less tool to be designed, coded, and verified.

In the absence of such tools, a decision may be made to develop such tools first, or to proceed without them. In either case, project costs will be increased greatly. For systems that will use minicomputers, software development may be a problem since, in general, minicomputers do not have as complete a battery of software-development support tools as large-scale computers. In addition, even with a reasonable battery of support tools, software development on a small-capacity computer is much less convenient than on a large machine for those applications which strain the resources of the minicomputer system. Therefore, an approach to the development of software for minicomputers that is often used in commercial organizations is to host the effort on a large-scale computer with the target as the minicomputer. This approach requires cross-assemblers, cross-linkage editors, and cross-compilers (if an HOL is to be used). This approach, while allowing greater flexibility, requires ready access to the cross capabilities for the particular host-to-target combination.

It should be pointed out that those host-to-target cross capabilities which are not available can be developed as a low-risk item with a cost of at most a few hundred-thousand dollars. In a minicomputer application, therefore, development of the support cross-targeting software, if it does not exist, might be cost-effective from an overall point of view. This would be true especially if a number of projects would realize benefit from the cross-targeting software.

The following lesson, learned from the commercial world, should be applied to the world of weapons-systems software development. Centralized software-development facilities should have available software-development and support tools, common routines, etc.* Such efforts alleviate the need to "reinvent" software, and, thus, reduce costs. Supporting these facilities should be an organization whose responsibilities are to acquire, verify, and maintain software-development and support tools. This would assure the visibility of such tools, and would eliminate redundant acquisition, verification, and maintenance efforts. Examples of such activities currently being implemented or underway in the DoD are: Navy System Design Laboratory (SDL), the Air Force SAMSO Aids Project, and the Ballistic Missile Defense Software Development System.

The DoD management policy as presented in Section 2.1 and the collection of available and relevant software-development support tools provide a basis for significant visibility and cost reduction in the area of systems development. The greatest savings in cost, with an attendant increase in reliability, can be realized by incorporating the management policy and support tools within a framework of a development methodology which encompasses the total weapon-system acquisition process--from requirements definition to deployment and maintenance. In the next section, several methodologies and support-tool collection efforts which could apply to various stages of the weapons-system acquisition process are discussed.

*Many of the commercially available tools and routines may be directly applicable to DoD problems.

2.3 Current System and Software Development Techniques and Methodologies

The following methodologies and techniques for software and systems development are presented in this section:

1. Structured Design
2. System Design Laboratory (SDL)
3. Information System Design and Optimization System (ISDOS)
4. Software Factory
5. Ballistic Missile Defense Software Development System (SDS)
6. Hierarchical and Input-Process-Output (HIPO)
7. Multipurpose User-Oriented Software Technology (MUST)
8. Dominic

These methodologies and techniques were selected for review because they were considered to be either fairly widespread or representative of approaches being developed. It is understood that these approaches may not represent the entire spectrum of software and system development techniques.

A separate section is devoted to each approach. In each of these sections, the salient features of the approach, as made available or published by the developing organization, are presented.

The last section focuses on shortcomings and a preferred approach applicable to the complete life-cycle of a military computer-based system.

2.3.1 Structured Design*

Structured Design is a methodology for system design currently in use at Hughes Aircraft Company. The methodology is basically the methodology developed by L. L. Constantine with modifications based on Hughes' real-time military experience. It consists of

* This section was supplied by (BEN76)

a set of guidelines and techniques which aid the engineer in conceptualization, decomposition and structuring of the system design. Two visual aids are used: structure charts and bubble charts.

The bubble chart shows the conceptual data flow of the system. It is composed of bubbles (circles) which represent the data transforms and connection lines which represent the flow of data. The bubble chart defines the required order of data transformation. It does not show control flow or indicate the modular structure. It is equivalent to a functional data flow with strict naming conventions.

The structure chart shows how the bubble chart is to be implemented. It shows the hierarchical relationship of the modules, the major procedural information, the table structure, and control/data flow. It is composed of boxes joined by connectors, and text with connectors indicating data and control flow.

The Design Process

The design process is divided into three phases. The first phase - the First Cut Design - is heavily conceptual. It seeks to identify the most basic tasks performed by the entire system, i.e., the "main mission" is delineated. All hardware, data system control, and module internals are ignored as the basic data flow and structure are established. The second phase - The Intermediate Design - is partially conceptual and partially real world. The data base, system control, etc., are given some consideration. The system decomposition process proceeds along prescribed guidelines. Independent sections are identified and the basic structure of the data base is discovered rather than assumed a priori. The result of this phase is the basic, or top-level design. The third phase - the Final Design - is primarily real world. Codeable modules are produced; the data base is finalized; the design is packaged for hardware; and coding documentation is produced.

Guidelines

A set of three categories of guidelines and a set of Rules-of-Thumb which are key to the design process and the creation of the bubble and structure charts are summarized below. The guidelines are many (about 40) and are very detailed.

- 1) Basic Guidelines - A set of formalized rules and methods. The basic guidelines deal specifically with the structured design methodology, i.e., modular decomposition: interactive use of the charts: and design considerations.
- 2) Secondary Guidelines - A set of quasi-formalized design methods. These guidelines attempt to describe the characteristics necessary in the human problem solving process and hence deal with concepts related to learning, understanding, and conceptualization.
- 3) Supplemental Guidelines - A set of non-formalized rules and methods which are derived from the basic and secondary guidelines. They identify diverse conclusions such as data base design, duplicate code, module compression, error handling, module size, and design changes.
- 4) Rules of Thumb - A set of eight very informal rules which are intended to increase the design speed and ease the burden on the designer. They cover such areas as naming, optimizing, and packaging.

Design Implementation

It is to be noted that structured design does not impose any implementation technique. That is, top-down, bottom-up, or most any other technique of implementation can be used.

2.3.2 System Design Laboratory*

System Design Laboratory (SDL) is a facility to provide designers and developers of embedded naval computer systems a comprehensive library of tools to design, model, implement and test systems. SDL utilizes the National Software Works (NSW) interface to access the tool library.

Conceptually SDL will offer tools to support hardware, firmware and software aspects of system design, modeling and development. The Initial Operating Capability (IOC), scheduled for 1 October 1976, will offer a full complement of software development tools for the Navy standard minicomputer, AN/UYK-20 and the Intel 8080 micro computer. These tools include high level language compilers and hardware simulator/emulator with interactive debugging aids. Expert personnel at NELC will provide users of SDL with both interactive assistance in specific tool use under NSW and documentation describing tool operations. User access to SDL is by interactive terminals using the EDATL TIP and by an Input/Output (I/O) station for printed listings and magnetic tape transmission.

During FY 1977, two tasks are scheduled for SDL. The first task will be to assist selected users in validating the IOC facility. This effort will establish operational data to insure SDL's immediate effectiveness and to provide requirements for future enhancements to SDL.

The second task will be to augment the SDL tool library. A key concept of SDL is extensibility; that particular capability to quickly and easily incorporate existing tools from other sources into the SDL tool library. Planned for 1977 addition are tools in the following areas:

1. Automatic partitioning of design
2. Automatic verification and validation for CMS-2
3. Multiple processor AN/UYK-20 emulator

* This section was supplied by (HAMP76)

Under investigation but not confirmed for implementation into SDL in FY77 are tools for these functions:

1. Problem Statement Language
2. Specification Language (AXES)
3. Modeling
4. Program documentation and maintenance aids
5. High Level Language

2.3.3 Information System Design and Optimization System (ISDOS)*

The ISDOS Project is being conducted by faculty and students in the Department of Industrial and Operations Engineering at the University of Michigan. The objective of the Project is to study the process by which Information Processing Systems are being built and operated. The term "Information Processing System" is used to mean a collection of hardware, hard software, programs, files and procedures which have been assembled to accomplish some requirements. Usually the basic requirements include the ability to produce outputs (answers to inquiries, reports, documents, messages, displays, etc.). Examples of such systems are various business data processing applications, information storage and retrieval systems, etc. The systems may include batch, remote job entry, on-line, interactive, real-time, etc., facilities or some combination. They may utilize data base management systems and communication systems. Most large organizations have a number of such Information Processing-Systems, frequently sharing hardware and files and occasionally also software.

The long term objective of the ISDOS Project is to develop methodologies for automating, as much as possible, the process of building systems. Shorter term objectives are to improve the present (primarily manual) systems building process and to prove feasibility by providing computer-aided methods.

* This section is excerpted from (TEI76).

One fundamental concept of the Project is that the basic data needed for building a system, namely the description of the requirements, should be recorded, as early in the process as possible, in machine readable form. Thereafter, the building of the system (construction of programs and data bases, etc.) is to be accomplished with the aid of the computer itself using algorithms which would analyze the requirements and aid in the construction of programs and data bases using operations research techniques to develop first feasible and then, eventually optimal solutions in accordance with stated performance criteria.

Computer-aided systems building will result in a number of benefits. Perhaps the most important is flexibility - a change in a requirement can more easily be incorporated into the system. Another major benefit will be will [sic] increase in productivity of systems analysts, designers and programmers since they will be concerned with the development and maintenance of the partially-automated system which will then be able to produce software to satisfy the user requirements. The computer, in effect, will be used to amplify the capability of analysts and designers. Even without full implementation, the formalization of the process can be used as the basis for education, research, and development.

The ISDOS System

The basic approach of the ISDOS Project is to focus on the system building process as an organizational activity similar to many other activities in that it depends very heavily on information flow and data processing. Furthermore, it is a process which is now primarily manual--in many organizations the only automated part being the compilation of source language statements into object code. The approach being followed in developing the ISDOS system is therefore very similar to that followed in automating other information systems:

- Study and describe the "present system."
- Improve the present system wherever possible.

- Propose a "new system" which makes use of the computer and operations research methodology whenever technically feasible and economically workable.
- Divide the proposed system into subsystems and develop a plan for phased development, testing, and installation of the subsystems.

In the development of the proposed system the "data base" approach is being followed. The steps are:

- Identify what information is, or should be, recorded.
- Develop a language for expressing this information.
- Provide a system for storing information in a computerized data base.
- Provide capability for displaying data with appropriate rearrangement, etc.
- Provide capability for checking for consistency, completeness, etc.
- Provide capability for analysis and evaluation.
- Provide decision-making aids.

The goal is to eventually replace the current documentation methods, in which documentation is prepared and analyzed manually, with one in which the necessary information is available for use by the various individuals who need access to it.

The initial subsystem is concerned with the phases of system building in which users requirements are determined, recorded, and analyzed and a "logical" system is designed. This requires the development of a language for stating requirements, and software for storing the requirements in computer manipulatable form for retrieval and analysis.

The methodology and the software developed in the ISDOS Project have been designed to overcome the factors which have limited the use of earlier attempts. A language, Problem Statement Language (PSL), to describe systems has been developed.

Problem Statement Language (PSL)

The requirements for a target system should be expressed in an unambiguous machine processable form. PSL has been designed to meet this goal by being able to accurately and completely express all relevant requirements for the logical design of the target system and by having a precise syntax and semantics. The complete collection of requirements for a target system written in PSL is termed a Problem Statement, in that it is a "problem" to be solved during the physical design and implementation phase.

A target system is described by listing its subsystems or components, by giving properties, and by stating relationships among the components. In PSL, each component is called an "object."

The relationships may be grouped into eight major groups on the basis of the "aspect" of the system which they describe. These eight major aspects are:

- System Input/Output Flow
- System Structure
- Data Structure
- Data Derivation
- System Size and Volume
- System Dynamics
- System Properties
- Project Management

In addition to the formal relationships, any information which is needed to describe an object and which cannot be specified by using one or more relationships can be specified in a narrative or text description called a comment entry. These comment entries are not named (as objects are named) and, therefore, apply to only one particular named object. A number of different types of comment entries may be defined depending on the type of object they pertain to.

The Problem Statement Analyzer (PSA)

As information about a particular system is obtained, it is expressed in PSL and entered into a data base using the Problem Statement Analyzer. The Analyzer is designed to operate in an interactive environment using the facilities of the host operating system. It is intended to be as system independent as possible. In order to achieve this independence, the software, including the data base management system is written almost entirely in FORTRAN IV and can be installed on any environment which has a FORTRAN IV compiler and sufficient memory.

The main subsystems are a Command Language Interface, Data Base Update Facility, Report Generator Facility and a Library Facility. The Command Language Interface module interprets commands in the Command Language from the user and causes the execution of the appropriate module to handle that command. The command processing modules fall into two categories: data base update modules and report generation modules. The command processing modules interface with the data base management system which is part of the Library Facility. The library facilities also perform certain peripheral functions, such as data base initialization, and dump and restore.

At any time standard outputs or reports may be produced on request. While this system is not directly oriented towards project management, a few reports, which are useful to project managers, are available. These reports include, in particular, the number of objects of each type in the data base together with the number and percentage which have specified properties. Such a data base summary can be used as a basis for evaluating progress made on the definition of objects, as well as used in estimating the size of the system being described.

There are also facilities for analysts to develop their own reports as they wish and include them in the standard report repertoire.

The reports can be used during the development activity, after the requirements are completed, and for maintenance throughout the life of the system.

During the requirements development activity, while data is being added to the data base and the data base is being modified, the analysts can use the various reports for making additions and changes. They can also produce reports to analyze various aspects of this system, and produce reports to determine what more information is needed. These reports can also be used by the Standards section to verify consistency on data naming conventions and use of other standards. The project leader can use the report to determine the complete extent of documentation to date.

After the requirements have been completed, the final documentation required by the organization can be produced semi-automatically, in a user standard format. Summary reports can also be produced to enable individuals to understand this system and reports can be produced which serve as specification for the succeeding activities.

During the system implementation and operation, changes will be suggested. The data base can be used to determine the effect of the changes, such as which other objects are involved, etc. The change in the evaluation procedure can be monitored by the use of the data base modification reports.

Summary of ISDOS

PSL/PSA is designed to be usable by any organization that develops information systems. Such an organization is incurring the direct cost of manual documentation and the hidden cost of correcting the mistakes in the system that are made as a result of ambiguity, inconsistencies and omission in the manual system. With PSL/PSA much of the manual and clerical work can be done by the computer.

PSL/PSA is designed to complement system development practices and procedures. It can be used regardless of which system development procedure is followed, which project management system is used or which documentation standards have been instituted.

2.3.4 Software Factory*

[Basic to the overall approach of Software Factory is a methodology which] emphasizes discipline and repeatability, but has sufficient generality and flexibility for application in a wide variety of situations. The methodology is not new or revolutionary. Most of its concepts have been used in successful software development projects. [It is intended that the system be applied consistently and be supported] with tools that will simplify and standardize application of the methodology. It is, then, the specific purpose of this [section] to describe this set of tools, which comprise the "Software Factory."

Structure and Components of the Factory

The Software Factory consists of an integrated and extensible facility of software development tools that supports a recommended methodology. The Factory is designed to operate on a host machine and use the facilities of the host operating system. The design is flexible so that new tools can be added as they become available, and the system components are written in higher-order language to enhance transferability of the system or its components to other machines. The structure and capabilities of this facility are, wherever possible, based on available technology to enhance near-term usefulness.

The following basic structural and control components of the system are either in operation or under development:

* This paper is excerpted from (BRA75).

The Factory Access and Control Executive (FACE), which performs control and status gathering services for all processors, supports the factory command language, integrates the processors with the system development data base, and provides program production library services.

The Integrated Management, Project Analysis, and Control Technique (IMPACT), which utilizes production data base information on milestones, tasks, resources, system components, and their relationships to provide schedule, resource computation, and status reports at the individual components level or summarized at any module or task hierarchy level. Since IMPACT is closely integrated with the development processors, much of the status information is derived automatically.

The Project Development Data Base, which is established for each project using the facility and contains all of the schedules, tasks, specification components, test cases, and their relationships along with the various forms of the evolving software components and the complete development status. The data base consists of two parts: a software development data base and a project control data base.

The development of program modules from their first functional definition, through the definition of their interface with other modules and system data, and finally - as developing source- and object-language programs - is reflected in the software development data base. It is an extension of the program production library concept.

The system and program descriptions and supporting management data are maintained in the project control data base. Early in the design phase, the management data is oriented toward the software system structure and the activities performed to develop the software system. This use of a project development data base furthers the automation of program development, management visibility, configuration control, and documentation.

FACE and the System Development Data Base, make up the framework which integrates and provides the control structure for the various Factory processors. As new processors are developed, additional commands can be added to the factory control language and new sections can be added to the data base.

The initial complement of Software Factory processors includes the following:

Automatic Documentation Tool (AUTODOC) - a tool to produce program and system documentation. It obtains a large amount of its information from sources in the system development data base that represent the actual state of the software system. The principle source of data for AUTODOC are the comments inserted into the program modules by the programmer. When a program module is defined during the design phase, it is entered in the system development data base in the form of specific AUTODOC comments. As the program module is further defined, the programmer adds source language statements. AUTODOC starts its operation by scanning a source module and extracting the special comments: it also reports on missing or incomplete comments. These special comments provide a means for the programmer to transmit information to various Software Factory development tools in a format that is compiler-independent and primarily language-independent. The special comments consist of a number of keywords, which indicate to the appropriate tool that information of a specific kind is contained in the comments which immediately follow.

Program Analysis and Test Host (PATH) - a program flow analyzer that analyzes a source program and inserts calls to a recording program at appropriate locations. The analysis of the program results in static profile analysis reports, which provide information about the structure of the program. During program execution, the recording program gathers data concerning the performance of the program for a given test data set: after program execution, the recorded data is processed and output. Rather than finding only

errors in a program, the objective of PATH is to quantitatively assess how thoroughly and rigorously a program has been tested and to support the improvement of the test design that will best satisfy the conditions of program verification.

Test Case Generator (TCG) - an automatic technique for the design of test data which, when input to the program under test, will provide the user with a means of fully executing all program statements successfully. The series of test cases that are generated will be stored in the system development data base. The TCG technique of determining the total network of statements in a program and of generating an adequate set of test cases has evolved from analysis of the work being done in the program verification and validation field.

Top-Down System Developer (TOPS) - a tool which provides a design verification capability. This involves the development of a modeling tool that provides not only the capability of describing and verifying a design, but also the facility to describe much of the control and data interface logic in the actual coding language.

This later capability provides early validation of a logic component that normally does not get checked until system integration. A further and most desirable aspect of the tool is a structure that allows the replacement of modeled system components with real components as they are implemented, thus allowing each new component to be validated in the context of the total system. Hence, this tool provides an automated approach to the "top-down implementation" technique in which succeeding levels of logic are implemented with calls to skeleton or simulated segments at the next lower level being used to close the logic loop. Thus, complete validation occurs at each logic level. Top-down implementation and testing by levels along with the accompanying exercise of actual interfaces, results in a continuous system integration process throughout the development cycle.

All of the tools described are further complemented by the existing utilities and interactive development capabilities of the IBM 370/VS2 operating system.

Summary of the Software Factory

The Software Factory is an integrated set of tools that supports the concepts of structured programming, top-down program development, and program production libraries, and incorporates hierarchically structured program modules as the basic unit of production. FACE and IMPACT record the hierarchical and interdependent structures of software system and programming tasks which are then used to evaluate project and program changes and to assist the system implementer in building the system and ensuring the completeness and accuracy of his designs and tests. IMPACT is intended to serve as an initial planning tool and as an integrated subsystem that supports software configuration management from the inception of a project through the post-delivery maintenance of a system. FACE and the system development data base are intended to provide a controlled environment for systems development and serve as a framework for a standardized approach to program implementation and verification.

The top-down phased description of performance specifications and system components and their relationships as required by IMPACT along with the consistent use of the various Factory processors provides discipline and uniformity to the development process, leading in turn to a high degree of repeatability with its consequent continuous improvement in proficiency.

The monitoring and control aspects of FACE and its linkage via the development data base with IMPACT provide objective development status visibility. TOPS and IMPACT provide earlier and more visible assessments of design completeness, and PATH provides a more quantitative assessment of testing completeness.

Since IMPACT supports a development data base in which each requirement is related to the system components by which it is implemented, the impact of requirement changes on the system architecture can be more completely and easily determined.

TOPS is a design support tool which provides more automation and better verification of the design process and also smooths the transition between the design and coding phases. PATH and TCG are verification tools which support the generation and performance assessment of program test cases.

While not specifically attacked by any currently-implemented Factory component, software reusability is enhanced by the careful system component structuring, the specific relationship with performance requirements, and the improved documentation inherent in software developed in the Factory.

The basic elements of the Software Factory (FACE, IMPACT, development data bases) are now operational and are undergoing some prototype use. It would be premature to assess the benefits which have accrued, but it does seem clear that a significant degree of discipline, organization, and visibility has been added to the development process.

It is intended that the Software Factory will be augmented by the continued development of more sophisticated tools and techniques such as application-oriented process design languages, reusability technology, correctness verifiers, and cross compilers, and will therefore evolve into a truly automated software development facility.

2.3.5 Ballistic Missile Defense (BMD) Software Development System (SDS)*

The large amount of money and research successfully spent in the SAFEGUARD and System Technology programs provided evidence that the ability to produce large complex real-time BMD systems was not keeping pace with the increasing complexity and capacity demanded of advanced defense concepts. In recognition of this problem, the Ballistic Missile Defense Research community initiated a software research program in the early 1970's to address the issues which had arisen in BMD software development. It became apparent early in this program that any attack upon a problem of this complexity must be of broad scope and yet provide a degree of formalism not currently available nor required for less stressing developments. This resulted in the Software Development System (SDS), which is a software development approach that forces early error detection, allows rapid assessment of design decisions, insures the capability to respond rapidly to change, and insures visibility and control of the development process. The program has concentrated upon developing a set of defined and measurable procedures supported by special purpose languages and advanced tools. This methodology has been developed, implemented, and is currently undergoing evaluation. It consists of a defined approach to Requirements Engineering, Software Design, Coding and Testing, and Verification and Validation. Component parts of the methodology have been used with favorable results. The integrated effect of the overall approach is currently under investigation through a series of proof-of-principle experiments.

The Development Cycle

[We will discuss the phases of the software development cycle which provide a framework within which the research activities have been pursued.] The initial activity, Data Processing System

* This section is excerpted from (DAV76)

Engineering, is concerned with the development and transformation of a set of system requirements into functional and performance requirements upon the data processing subsystem which will insure proper system performance. Included are the activities of system design, subsystem definition, interface specification, performance allocation to the data processing subsystem, and identification of normal and contingency system operating rules. This activity would proceed until the data processing subsystem has been identified by its functional and performance requirements and interface definitions. The resulting requirements (Data Processing Subsystem Performance Requirements - DPSPR) are communicated to the data processing subsystem developers for further decomposition, design, implementation and testing.

Based upon the DPSPR, detailed subsystem requirements are developed for the data processing subsystem in a phase known as Requirements Engineering. In the Requirements Engineering phase the detailed subsystem computational requirements are developed in a Software Requirements Engineering (SRE) activity which forms the basis for data processing hardware requirements. The Software Requirements Engineering activity is a process of iterative addition of design detail with the emphasis upon avoiding unnecessary constraints upon the following process design phase. This activity may include the demonstration of the feasibility of requirements through the development of non-real-time simulation. It should be noted that the term requirements used in this activity is the statement of requirements upon processing to meet system objectives, e.g., the timing accuracy and computation requirements for the tracking of a potential target, as opposed to the formal definition of detailed requirements for software modules. Design decisions required in developing computational and functional requirements are those affecting the functional and logical flow of requirements, e.g., a decision to specify synchronous or asynchronous tracking. The requirements are also developed with a minimum number of decisions affecting processor configuration, core size, etc., with

the resulting Process Performance Requirements (PPR) minimizing any optimization for a particular hardware configuration. [Since a highly sensitive real-time system is involved], the response time for various computational processes are very critical. The PPR contains performance requirements for each computational path within the process including interface definitions, suggested candidate algorithms, etc., required to meet system performance.

The hardware requirement aspects of Requirements Engineering are concerned with the development of data processing hardware requirements which will insure the selected configuration will satisfy and meet system computational requirements of the system. Decisions to be made in this phase include specifications of the various hardware characteristics, such as size, number of processors, performance, etc.

The PPR functional and performance requirements are then analyzed together with the characteristics of the hardware system. This mapping of requirements with the hardware characteristics form the first step of Process Design Engineering. This results in a top level software design and a definition of operating system requirements. From this structure the design proceeds in a structured manner with implementation and testing of each level of definition.

At each step of the SDS methodology, a comprehensive testing procedure, Hierarchical Verification and Validation is used to validate and verify the functional and/or performance characteristics of the software phase. Total system requirements and the specifications derived from them are used as the absolute reference to verify the design and implementation correctness at each level.

Software Requirements Engineering Methodology (SREM)

The approach to the development of SREM concentrated heavily upon definition of its requirements in the early development and has proceeded as a sequential set of proposed approaches backed by empirical verification. The resulting SREM consists of a combination of

languages, tools and procedures which will reduce or eliminate known error sources. As an example, the generation of testable requirements in SREM is addressed in a positive fashion through the development of a structure for statement of requirements which allows the identification of (1) test points in the requirements, (2) identifies the data to be collected at these points, and (3) an executable description of the tests to which the data will be subjected. The requirements statement thus contains a clear description of the tests to which the software will be subjected. Computer aided Simulation Generation Techniques are also an integral part of the methodology and provide the capability to validate, statically and dynamically, the requirements description. Another example of a positive approach for error reduction are the techniques used to reduce errors, delays and frustrations involved in documentation. This was addressed by developing a set of automated documentation tools based upon the ability to flexibly access information stored in a requirements data base.

Positive approaches in SREM to insure traceability of requirements are contained in features of the Requirements Statement Language, RSL, which have elements TRACES FROM, TRACE TO to insure upward and downward traceability of requirements. In addition, the language element DECISION allows requirements affected by a design decision to be later traced to that decision. Flexible data extraction of requirements information stored in a relational data base provides a rapid and reliable means to insure location of all affected requirements. Accurate modification of requirements is insured through the ability to rapidly enter data, interactively or batch, tools which will check for consistency with existing descriptions, and static and dynamic verification through simulation.

Requirements Statement Language (RSL)

Requirements have been stated in a wide variety of languages ranging from English text, equations, logical expressions or to machine processable forms such as PSL. Large projects tend to have a predominance of English text with its associated ambiguities and misinterpretations. While some work and progress has been made in the area

of machine recognition and analysis of English this is for the most part manually resolved through review meetings, interface control boards, etc., in a lengthy and costly manner. To avoid ambiguity and promote precision and communication of BMD requirements, the Requirements Statement Language (RSL) was developed. RSL supports the statement and documentation of requirements, and along with the Requirements Engineering and Validation System provides a flexible means for statement, verification and documentation of requirements. As was previously mentioned the desire is to provide for naturalness of expression, unambiguous communication and machine processing. The language requires a minimum of punctuation and when passed through a post processor produces a product very difficult to distinguish from English except for the repetitive nature of the statements.

Support Software

The BMD systems which are being described are so large that it is virtually impossible for humans to ensure that all parts of a requirements specification are complete, consistent, and correct. The rigor and thoroughness of the computer is a great asset in checking requirements specifications. A computer-aided system must enforce some measure of discipline on the creativity of the engineer so that the development process always moves in the direction of reduced ambiguity and increased consistency. For example, the computer can perform static checking of the requirements to illuminate inconsistencies such as conflicting names, improper sequences of processing steps, and conflicting uses of items of information which must be present in the system. With a flow-orientation such as the one that we have developed, the computer can even check the dynamic consistency of the system through the use of a simulation. The set of support tools is referred to as the Requirements Engineering and Validation System (REVS).

Process Design Engineering

The BMD approach to the design coding and testing of software is encompassed in the Process Design Methodology, PDM. This approach consists of a set of process design procedures and techniques supported by software development tools accessed through a single language PDL. PDM consists of the following major components:

- A structuring technology to allow an unambiguous and traceable transformation from computer-independent computational requirements (the PPR) to a top-level process design effectively.
- Design, implementation, and testing techniques supported by PDL.
- A Process Design System, consisting of support tools for automating such functions as requirements traceability, configuration management, library management, simulation control, data collection and analysis, and documentation.
- A set of models and techniques to accurately estimate project costs and schedule. These approaches provide information to assist management in the effective control of BMD software development.

Initial structuring techniques identify a set of tasks and hardware/software trade-offs that must be performed to identify a top level software structure. The Process Design System, through the capabilities of PDL2 and its support software, provide the tools to make the trade-offs and support evolution to the final code. These trade-offs not only include the tasking structure but also impact the operating system design.

The design of the software proceeds in an evolutionary manner as each task is further partitioned into a set of computational algorithms that are executed via a prescribed sequencing logic. This approach proceeds in a somewhat top-down manner, seeking to initially describe the sequencing logic and final process and adding detail through increasingly detailed algorithmic models. The process is thus represented as a mixture of modules at each stage of the development where some modules are detailed code and some are merely skeletons of the tasks to be performed. The Process Design System keeps track of these modules as they evolve through the development steps into final code, executes the process with an environment simulator at any stage of development, and analyzes the performance of the process. Since the entire process is represented at any development stage, each module is tested in its complete environment. The interface problems usually associated with software integration are identified early in the design where they can be more easily resolved.

This approach is designed to force resolution of control, structure, and interface problems early in the design cycle prior to the major coding efforts. The integration of more detailed algorithms in a logical "forward integration" sequence, in which the highest level of the system is detailed first, and subsequent processing steps are detailed in the order of processing. In this way, analytic data provided by the initial processing steps is available as an input to subsequent analytic algorithms. Iteration of this implementation, test, and evaluation cycle results in the complete real-time process.

Throughout the development cycle, the evolving process is tested against the performance criteria stated in the Process Performance Requirements to ensure that the real-time software will satisfy the system requirements.

The PDS capabilities are provided through ... [a set of] integrated... software tools.

Verification and Validation

The testing of large interactive, real-time systems such as BMD provides a set of problems which strongly parallel those of the development of the tactical code. A testing philosophy which provides the maximum assurance of early error detection is essential to provide a highly reliable software product. Significant investigations into testing via non-real-time interactive and externally resident simulation of all non-data processing components called a System, Environment and Threat Simulator (SETS) discusses an approach to providing a significant decrease in the testing and cost of BMD system testing as well as providing the ability to better define the "performance space" of the software under test and determine a measure of its robustness over wide ranges of input. This is being approached by investigating automated techniques for the interactive generation of test cases, performance evaluation techniques to allow the assessment of the BMD software response to that threat, and algorithms to intelligently perturb the input space.

Preliminary Research results based upon experience with SETS has shown the interactive construction and modification of test cases does provide significant reduction in the time to develop test variants. Performance measure characterization to all assessment of the software response to the threat still suffer from the ability to relate data processor subsystem functions to higher level system functions. This activity will result in a feasibility demonstration of achieved capabilities early in 1977.

2.3.6 Hierarchy plus Input-Process-Output (HIPO)*

Some IBM personnel believed that programming systems documentation emphasizing function could contribute to the efficiency of the program maintenance effort by speeding the location in the code

*This section is excerpted from (IBM).

of a function to be modified. They developed the HIPO technique of documenting function to meet this objective. Today HIPO is being used by some groups throughout the development cycle as a design aid and documentation technique.

Hierarchy plus Input-Process-Output (HIPO) addresses the requirements of the people who rely on documentation for many different purposes. A manager or user, for example, may want to obtain an overview of the system. An application programmer needs the documentation to determine program functions for coding purposes. Someone involved in a maintenance activity requires documentation that quickly identifies functions in which changes have to be made. HIPO meets these needs because of its graphical representation of function, its organized nesting of increasing detail, and the depiction of input and output data items at each level.

A HIPO package consists of a set of diagrams that graphically describe function from the general to the detail level. Initially, each major function is identified and then subdivided into lower-level functions; the summation of the lower-level functions equates to the higher-level functions. Programs are then developed starting with the functions described in the topmost level of diagrams. HIPO diagrams can be used from the start of the project through implementation and are useful for program maintenance by easing the identification of the code to be changed.

The major objectives of HIPO as a design and documentation technique are to:

- Provide a structure by which the functions of a system can be understood. The diagrams are organized in a hierarchy structure (see Fig. [2.3.6.]), much like an organization chart, where each diagram at any level is a subset of the level above it. Complex systems or programs can thereby be broken into manageable pieces. For example consider a project of mapping the United States. If the project team developed page after page of prose describing the states and highways, and each city's streets and all the

connections, it would be difficult to verify or use this text even if one had the time. Even if all highways in the states and all streets of all cities were shown graphically on one diagram, it would be impossible to work with all the information at one level of detail. Therefore, a hierarchical scheme of maps, some showing states with highways, others showing cities with streets, best allows a person to view the total network as required.

- State the functions to be accomplished by the program rather than specify the program statements to be used to perform the functions. A section in the diagrams called "Extended Description" provides additional information about the functions to reduce reliance on other documentation and to provide guidance to programmers.
- Provide a visual description of input to be used and output produced by each function for each level of diagram (see Figure 2.3.6.1b). Typically, the most important objective in a programming system is to produce output that is technically correct and meets users' requirements. HIPO allows this transformation of input data to output data to be visible.

Kinds of Diagrams in a HIPO Package

1. Visual table of contents - This diagram contains the names and identification numbers of all the overview and detail HIPO diagrams in the package and shows the structure of the diagram package and relationship of the functions in a hierarchical fashion as depicted in Fig. [2.3.6.1a]. It also contains a legend indicating how symbols in the package are to be used. With the visual table of contents, the reader can locate a particular level of information or a specific diagram without thumbing through the entire package.
2. Overview diagrams - High-level HIPO diagrams, called overview diagrams, describe the major functions and reference the detail diagrams needed to expand the functions to sufficient detail.

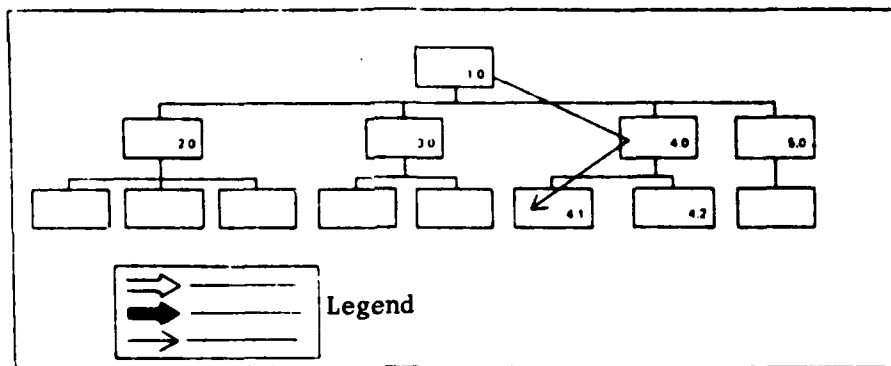
The overview diagrams provide, in general terms, the inputs, processes, and outputs. The process section contains a series of numbered steps that describe the function being performed. The input section contains those data items used by the process steps. Arrows connect the input data items to the process steps. The output section contains those data items that are created or modified by the process steps. Arrows connect the process steps to the output data items. An extended description area can amplify the process steps and input and output data items. The extended description also refers to lower-level HIPO diagrams, non-HIPO documentation, and code. Figure [2.3.6.1b] is an example of an overview diagram.

3. Detail diagrams - Lower-level HIPO diagrams contain the fundamental elements of the package. They describe the specific functions, show specific input and output items, and refer to other detail diagrams. The detail diagrams contain an extended description section that amplifies the process steps and references the code associated with the process steps. They also reference other HIPO diagrams as well as non-HIPO documentation such as flowcharts or decision tables of particularly complicated logic, record layouts, and so forth. The number of levels of detail diagrams is determined by the number of functional subassemblies, the complexity of the material, and the amount of information to be documented. Figure [2.3.6.1c] is a sample detail diagram with an extended description section.

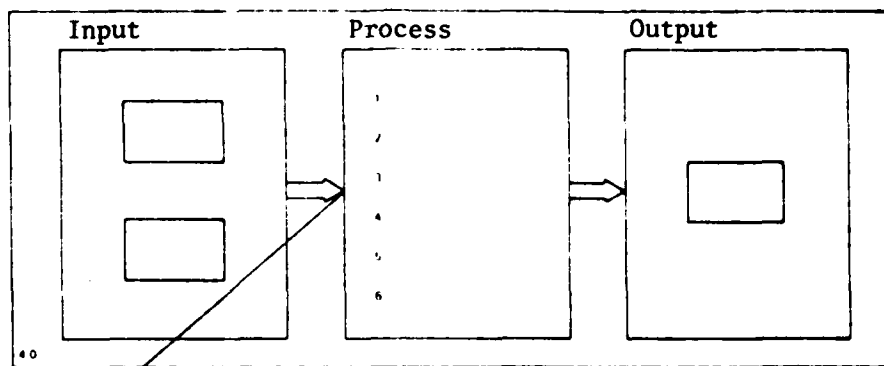
How HIPO Fits with Other Improved Programming Technologies

HIPO assumes that a system (a collection of related programs) will be organized into a hierarchical structure of functions. The scope of the topmost function encompasses all subfunctions. Those subfunctions that require further clarification are treated as major functions consisting of additional subfunctions. This process continues for as many levels as required until all functions are defined.

(a) A Visual Table of Contents



(b) Overview Diagrams



(c) Detail Diagrams

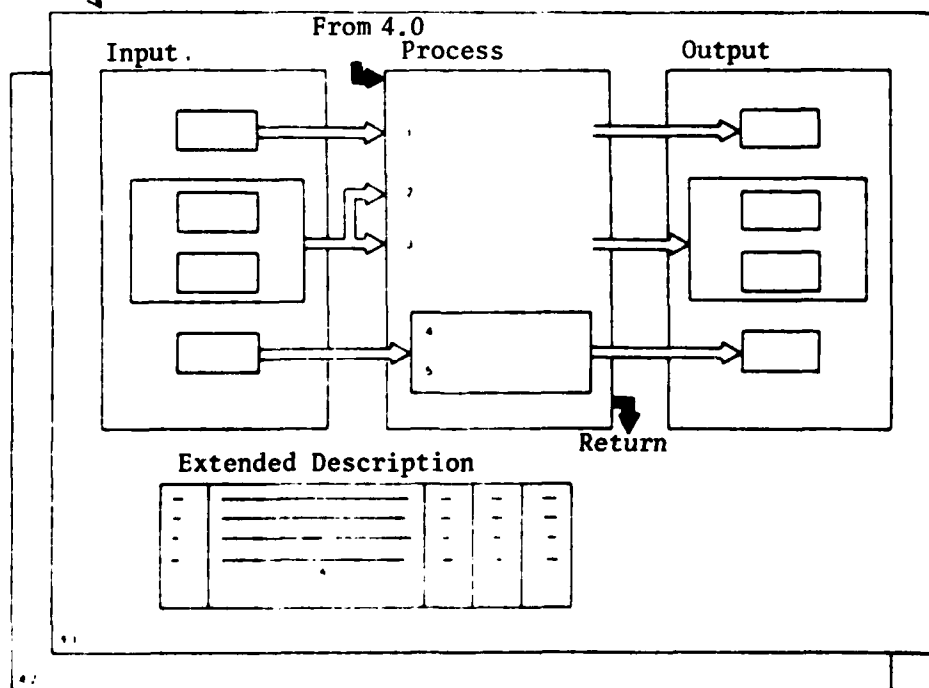


Figure 2.3.6.1: The HIPO Technique

The hierarchical structure of HIPO is well suited to a functional design made by starting at the top and subdividing into increasingly lower levels of detail. In top-down development, the functions are implemented in the same sequence as this structure. The top module contains the highest level of control logic and decisions for each program within the system, and either passes control to lower-level modules or identifies lower-level modules for inline inclusion. The parts of each program are continually being integrated.

If structured programming is used in the implementation, the functions are considered as single entities. The code is written in segments, each with a single entry and single exit. These segments can be created from HIPO diagrams drawn with only one entry and one exit. Some recommended practices to use in writing structured code are to limit a segment of code to one page (approximately 50 lines), and to indent subfunctions or substructures. These practices enhance readability and allow easy understanding.

Combining top-down development and structured programming results in a program of extreme modularity both in function and logical structure. HIPO diagrams are a logical extension of the functions identified in top-down development and provide the necessary documentation from the start of a project through implementation.

Another concept that is being used with top-down development and structured programming is the chief programmer team organization. Team operations represent a change in approach from a loosely structured group of programmers to highly structured team of programming specialists. The nucleus of a team operation consists of a chief programmer, a backup programmer, and a librarian; other members are included as required. The chief programmer designs the major functions of a system and codes the topmost levels of modules. He is assisted in these tasks by the backup programmer. The librarian is responsible for entering all information into the development support library, the principal objective of which is to provide

constantly up-to-date and visible descriptions of the programs, test data, and exact status of the system under development. The HIPO documentation should be included in this library and may be maintained by the librarian. When other team members are added to develop and program low-level modules, they can work independently down various paths of the hierarchy with separate HIPO diagrams.

2.3.7 Multipurpose User-Oriented Software Technology (MUST)*

Problems that result from the fundamental interactions between program, computer, and programmers, and the traumatic and costly effects when changes are made between them are resolved through Multipurpose User-Oriented Software Technology (MUST). MUST provides a more effective user interface so that the programmer-user, the system designer, and the engineer responsible for the flight control system can interact with the flight computer. The user-oriented system contains the software that the user needs to effectively interface his problem to his machine.

First, the user is provided with a requirement definition/analysis tool which permits him to express his system design requirements in a precise problem statement language and then analyze the software requirements to meet those system specifications. He then has a compiler-writing system (CWS) which permits the effective utilization of HOL programming procedures. The designer has less need for a special programmer if he can understand both problem and machine. A programmer can use the HOL to produce code that is more problem-oriented, more easily understood by the system designer, more easily documented, coded more rapidly, and more easily maintained or transferred. The programmer-user is aided by a modular library which has many well documented typical flight control elements available in standard HOL software so that changes can easily be accomplished.

*This section is based on information in (STR76).

The CWS allows code to be generated for both target and host machine. On the host machine the programmer-user can check his algorithm by deriving solutions and by using verification and validation tools to thoroughly check the performance of his program. For example, he could determine program flow, whether all variables were initialized before use or that none were out of range if all parts of the code were used, and which parts of code were most extensively used if further optimization is required. MUST, therefore, would provide the programmer many tools for checking and debugging his software.

The CWS can, using the exact same HOL program, develop code for the target machine. Hence after debugging, the actual load tape for the machine in relocatable binary format modules could be obtained. Linking a program with other available machine language programs could then be accomplished. If necessary, the programmer-user can write critical procedures in the machine language. Besides running on the actual machine, an interpretive computer simulator (ICS) system in MUST can be used. Here the host-computer generates the bit pattern in the registers, and arithmetic units and the actual values in the memory locations resulting from the execution of each machine instruction. Hence, the programmer-user can solve any remaining problem by direct execution of code as well as determine the precise machine instruction time cycles used.

SPECIFIC MUST DEVELOPMENTS

The broad objectives of the program will be accomplished by developing and demonstrating technology advances in the flight software development process by specific technical goals and targets enumerated below:

Support Tool Development

Software support tools are used to relieve the programmer-user of many of the tedious and repetitive tasks associated with software

translation and checking. They also provide him with clear insight into the program flow and execution.

Compiler Writing System - (CWS)

Target: To develop a user-oriented, compiler-writing system covering a variety of source languages and target computers where utilization requires detailed knowledge of the desired language and the target computer system idiosyncrasies rather than of sophisticated compiler writing principles. To demonstrate the order of cost savings, lead-time reduction, and inherent reliability achieved through compiler-writing techniques as applied to a wide range of flight computers and research flight project requirements.

Assembly Language Generations

Target: Develop or modify an existing generalized (universal) assembly language program for inclusion into the MUST system as a complement to the CWS and for those conditions where assembly language programming is needed.

Static Code Test Tools

Specific Targets: The development of MUST compatible proof-of-correctness and code anomaly analysis tools will be pursued. In addition, a MUST compatible sneak analysis tool will be developed.

Dynamic Code Test Tools

Specific Targets: Develop the capacity to automatically aid in the generation of test cases including assessment of test coverage and range variables; to model the flight computer using a flexible ICS to find errors, trace program execution, and determine software performance. The dynamic test tool will include the extensions of dynamic test tools such as RXVP to allow MUST compatibility.

Modular Adaptable Software Library

Target: Define and develop a modular software library and user interface for both aircraft and spacecraft applications. Develop techniques for the checkout and testing of library software on a modular basis to reduce implementation costs for large flight computer programs.

Requirements Definition Analysis

Target: Part of the MUST system will be a tool to aid in the unambiguous, precise, and complete statement of the software requirements. The tool will allow the user to detect inconsistency between the inputs and outputs of the various elements of the software being designed.

Integrated Flight Software Development System

Target: Investigate and implement software support tools with multilanguage capability. Define and develop an interactive interface involving user-oriented software specification design and test utilities under MUST.

Languages for Distributed Compiler and Microprocessor Systems

Target: Define the impact of microprocessors and distributed computing systems on the flight software development problem. Determine the features necessary to effectively describe the computer executive processing structure and its software implementation and integrate these features into MUST.

2.3.9 Domonic*

The purpose of the system is to help document, monitor, and control software development projects. It is written in COBOL which makes the system machine independent. It can operate in either a batch or interactive mode which makes it easily adaptable to a user's style of development. It can monitor all phases of software development projects. Data can be gathered by the monitor which is useful in categorizing errors, predicting programmer productivity, and validating software reliability models.

Furthermore, the system will be applied to monitor development projects and to gather programmer productivity data and program error data. The productivity data will be used to develop an accurate programmer productivity model. This model will be useful for accurately estimating project cost and completion times. Program error data will be useful in validating existing software reliability models and for developing new models.

Technical Objective

It is expected that a system of digital programs for the IBM 360, UNIVAC, and CDC computers will be produced. This system will be capable to document, measure, improve and predict the quality and reliability of other digital programs. Furthermore, the system will be capable to evaluate software projects which have been developed without using this system. Major modules from the system would be configured to create the project evaluation system. The system would be able to: 1) predict software reliability using software acceptance reliability model aids, 2) drive software packages that automatically test software, 3) analyze program structure to determine if structured programming techniques were used, 4) check program and appropriate documentation to determine if local programming practices have been followed, 5) check program to determine if proper standards were used.

* This section is excerpted from information supplied by (DAM76).

2.3.9 Summary

The previous sections have presented various approaches to solving some of the problems of systems development discussed early in this chapter. These approaches range in stage of development from those which are now undergoing conceptual definition to those currently in use.

Most of the approaches currently available intentionally concentrate on a subset of the spectrum of problem areas confronting systems developers. Of the currently developed approaches discussed in this section, the BMD Software Development System is the most comprehensive with respect to its applicability across the development spectrum. The BMD approach, however, utilizes different methodologies in the different phases of development which results in the requirement to "shift gears" as the development progresses through the various phases. Consequently, interfaces between phases may present problems.

All of the approaches presented require manual intervention in the various development phases (e.g. by the designer, implementer, manager, etc.). Such manual efforts are primary sources of errors in systems development; this is supported by the large effort (and expense) that has been required to verify these manual processes in the various approaches.

In order for reliable, cost-effective systems to be developed in an efficient, straightforward manner, the systems development approach must have:

1. A methodology which can be applied consistently throughout all phases of the development process.
2. A methodology which has a formal basis which will allow elimination of a major portion of the verification required to prove that the system developed meets its intent.

3. Automated tools which enable the system development to proceed automatically with a minimum of manual intervention, from the requirements to the deployed software.

The approaches presented in the previous sections of this chapter are, without question, improvements over earlier systems development approaches. They do not contain, however, the required methodology or support tools for an integrated approach in systems development.

3.0 RATIONALE FOR ISDS/HOS

3.0 RATIONALE FOR ISDS/HOS

3.1 Background

Higher Order Software (HOS) evolved from attempts to bring engineering rigor and discipline to the various phases of computer system development (HAM73a). The natural evolution of these separate efforts on the distinct phases of system development has resulted in a formal methodology that is applicable to the entire system design and development process (HAM73b). In the following paragraphs, the background and early evolution of HOS is outlined.

The Apollo study was of great value for determining the direction for future software efforts (HAM73c) (HAM71). For example, the fact that 44% of all of the anomalies in the software were found by "eyeballing" was a clear indication that static verification was important. Also, the fact that seventy-three percent of the anomalies studied occurred at software-to-software interfaces encouraged concentration on interface correctness. Another realization resulting from this software analysis was that flexible software systems are a key to managing software developments. A case in point is the Apollo on-board asynchronous systems software. If the Apollo Guidance Computer (AGC) systems software had not been asynchronous, the development process would have been much more expensive, much longer, and at least one of the Apollo flights would have been a disaster (HAM72). (During the aforementioned APOLLO flight, the system was required to reconfigure its job queue to avoid overload.)

The difficulties encountered during integration at the Assembly Control Supervisor (ACS) focal point in the Apollo development led to the requirement to formally define software modules (HAM76a). Programmers were required to deliver finished modules to the ACS

who was charged with ensuring that each module:

- a. performed its specified function
- b. interfaced correctly with other modules
- c. did not impact other modules in an unreasonable way.

It was observed that modules which were defined with standard interfaces produced fewer problems for the ACS than modules defined in non-standard terms. Further, from the management viewpoint, the ACS concept was found to be beneficial in that it established a focal point through which all official software was filtered, thereby providing increased management visibility and software integrity.

Over the last several years, a design approach has evolved wherein large problems are divided into smaller problems, each of which is defined as a new problem (DAH72), (JAC76), (MIL71), (MYE74), (ROS76), (SNO72), (STE74), (WIR72), (YOU75). The division process continues until the problems are small enough to solve. Hopefully, the aggregation of the solutions to the small problems solves the original problem. In order for this to work, it was found to be necessary to decompose the problem so that the individual solutions fit together properly.

During the APOLLO software development, anomalies were analyzed and characterized to determine rules which would have prevented such classes of anomalies. Checklists for all of the disciplines of the software development process (design, implementation, verification, documentation, and management) were refined (HAM73b). The refined manual checklists proved to be identical for each discipline. Checklist items were then categorized, and it was determined that many of the manual processes could be automated.

3.2 Concept of the HOS Formalized Approach

The key features of the Higher Order Software Methodology were derived as specific solutions to the problem areas discussed in Section 3.1. To address the problem of interface errors, six axioms were developed, the adherence to which will assure interface correctness without requiring program execution. These six axioms, which became the base of the formalized system, explicitly define hierarchical software control. These axioms distinguish HOS from other software methodologies.

In order to assure software reconfigurability in real-time, an asynchronous approach was adopted in which processes do not have to fit into timed intervals. Through the use of the HOS methodology, such processes can be dynamically reordered since the systems software can schedule processes based on priorities (clocks or other events).

The observation that modules should be formally defined in order to provide more efficient module integration lead to the HOS approach to modularity. As a result, the definition of an HOS module includes in a formal way its functions and control aspects.

The Assembly Control Supervisor concept has shown the importance of a management structure in developing reliable software. Consequently, the HOS concept also places constraints on a management structure by requiring its adherence to the HOS axioms. Through this approach, the following management benefits accrue: (1) the methodology is applicable to systems over the total range of sizes and complexities; (2) there is reduced dependence on individual people; (3) clear traceability of project progress is possible; (4) the impact of a manager's decision on other managers is clearly defined; and (5) the management structure helps to determine the relationship between groups and organizations.

In the area of problem decomposition, HOS formalizes the process of dividing problems to ensure that the interfaces between parts of the solution are correct. The decomposition is based on the six HOS axioms which results in a tree structure solution to the problem where the nodes of the tree represent hierarchical levels of specification.

Through the formalized approach to software development offered by HOS, a set of techniques and automated tools applied in an integrated manner to the various software phases and disciplines, has been defined. Such tools will enforce a rigorous software engineering approach, the consequences of which will be reduced software costs and much greater software reliability.

The rationale delineated above is further supported by the MITRE and APL studies referenced in Chapter 2. Specifically, the MITRE study identified the "lack of discipline and engineering rigor applied to the weapons systems software acquisition activities" as the major contributing factor to the problem of controlling increasing costs and improving the quality of software in weapon systems. The APL study agreed that "the lack of systems engineering methodology to computer systems design is at the root of a number of critical problems in the development of major weapons systems." HOS forms the basis of the Army's Integrated Software Development System (ISDS/HOS) which is shown in this report to be a formal systems engineering methodology with supporting software tools for developing reliable tactical real-time software.

4.0 FOUNDATIONS OF ISDS/HOS

4.0 FOUNDATIONS OF ISDS/HOS

4.1 Preliminaries

4.1.1 Trees and Functions

Using the ISDS/HOS approach, software systems can be developed with the aid of simple mathematical concepts and a set of software engineering axioms. In this section, the required mathematical concepts are described. In the following section, these concepts will become the means (language) by which ISDS/HOS is described.

The two mathematical concepts required in order to describe ISDS/HOS are the tree and the function. The tree is a structure comprised of a finite number of nodes which are connected by branches as shown in Figure 4.1.1.1.

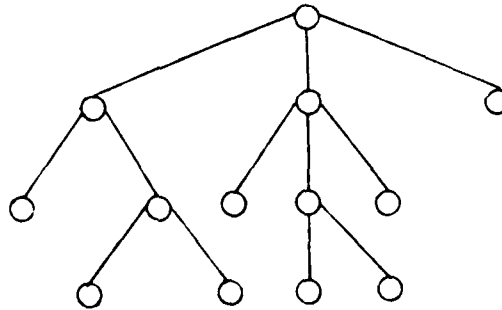


Figure 4.1.1.1
An Example of a Tree Structure

A branch may be interpreted as entering a node (from above the node) or leaving a node (from below). The unique node at the top of the tree that has no branches entering it is called the root of the tree. A node that has no branches leaving it is called a leaf of the tree. It should be noted

that all nodes other than the root have exactly one entering branch.

A root is considered to be at level 0 of the tree (see Figure 4.1.1.2). As one starts at the root and traverses a path to a leaf, each successive node defines the next level

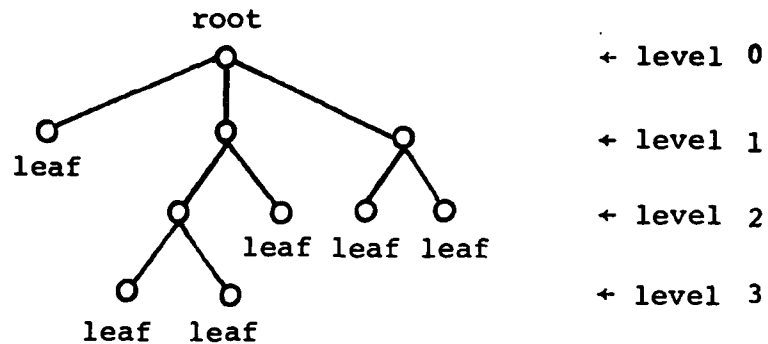


Figure 4.1.1.2
Tree Levels

of the tree. If a branch leaves node A (Figure 4.1.1.3) and enters node B, then node A is the parent of node B, and node B is an offspring of node A. (In Figure 4.1.1.3 node C is also an offspring of node A.)

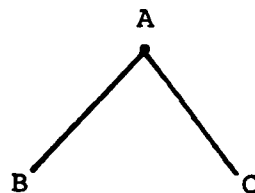


Figure 4.1.1.3
Parent-Offspring Relationship

A nodal family is a particular parent node and all of its offspring (see Figure 4.1.1.4).

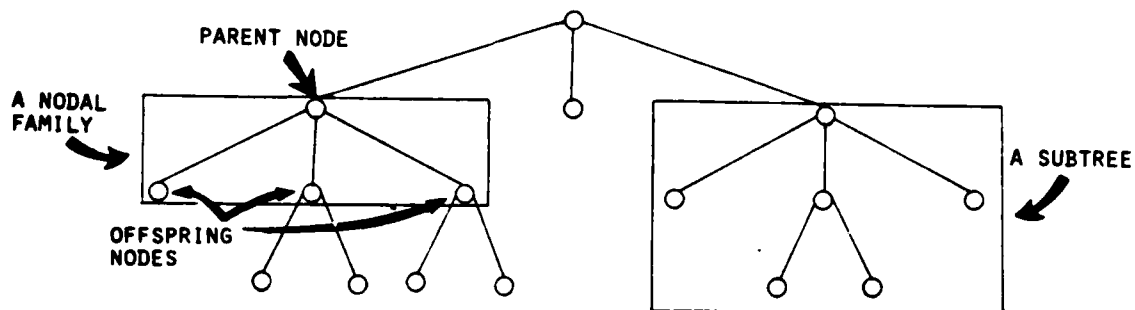


Figure 4.1.1.4
Tree Substructures

If there exists a sequence of nodes n_1, n_2, \dots, n_k , such that for every i , n_{i+1} is an offspring of n_i , then each n_{i+1} is a descendant of n_1 . A particular parent node of the tree together with all of its descendants and connecting branches is the subtree defined by the given parent.

If α and β are set elements (from either the same or different sets), then (α, β) denotes the ordered pair consisting of α and β in that order. (Thus, the ordered pair (β, α) is not the same as (α, β) except for the case where α and β are the same elements.)

If two sets, X and Y , are given, and x and y represent arbitrary elements of X and Y , respectively (i.e., x and y are variables), then any set of ordered pairs of the form (x,y) is a relation between X and Y . For example, if $X = \{1,2,3,4,5,6\}$ and $Y = \{m,s,e,w\}$, then one possible relation between X and Y is $R = \{(4,m), (3,s), (4,w)\}$.

The set of left elements of the relation is called the domain, and the set of right elements, the range. In the above example, the domain is $\{3,4\}$, and the range is $\{m,s,w\}$.

A relation is a function when each element of the domain has only one corresponding range element. If f is a relation between X and Y , and f is also a function, then we say that " f is a function from X into Y " (usually written $y = f(x)$). An example of a function is

$$f = \{(1,m), (2,s), (4,m), (6,e)\}$$

as illustrated in Figure 4.1.1.5.

In the sections that follow, the variable that represents the domain elements is referred to as the input variable, and the variable that represents the range elements is referred to as the output variable. Individual domain and range elements may be called inputs and outputs, respectively.

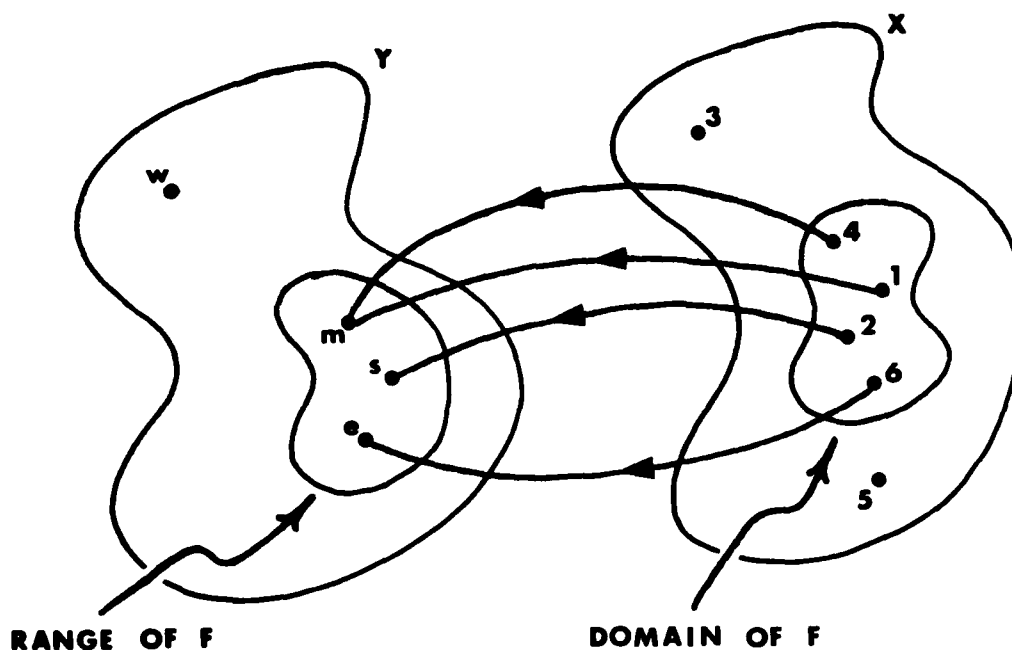


Figure 4.1.1.5
Illustration of a Function from X into Y

4.1.2 Modules and Nodal Families

In ISDS/HOS, the decomposition process for a system results in a tree structure. At the start of the decomposition process, the entire system is represented by the root of the tree which, hopefully, represents the requirements for the system. This description, however, has many implicit (hidden) requirements. In order to explicitly arrive at the complete description of the requirements of the system, the root is decomposed by replacing it by a nodal family, which represents the decomposition of the root. This decomposition process, that of replacing a function by its nodal family, can be continued until the entire system has

been explicitly specified to whatever detail is required or desired. It may turn out that during the decomposition process, a requirement is shown to be erroneous or missing. In such a case, an iteration of the system description is required.

The parent node of the nodal family controls its offspring. When referring to this control relationship, the parent node will be called a module, and its offspring will be called functions. The offspring of the nodal family are the functions required to perform the module's corresponding function (MCF) (i.e., the function that the nodal family replaces).

The resulting tree represents the system where the leaves represent, in an abstract machine sense, the machine "instructions" that are to be actually performed; the intermediate nodes represent control with respect to the performance of these leaves. It can be shown that the ISDS/HOS axioms provide rules for the way that a nodal family can be constructed. These methods for constructing a nodal family will be presented after the axioms are introduced.

4.2 The Axioms

Axiom 1 is concerned with invocation which is the act the module carries out in order to set up the initiation of the execution of its function. The axiom limits this right of invocation so that the module, as a parent in a nodal family, can only invoke its offspring. Thus, the module (1) cannot invoke itself, (2) cannot invoke its parent, (3) cannot invoke any of its descendants other than its offspring, (4) cannot invoke another offspring of its own parent, and (5) cannot invoke another parent's offspring.

Axiom 2 is concerned with the function associated with the module. For any given element in the domain of the module's function, the module is responsible for producing the correct corresponding range element. In other words, the job of the module is to perform a function. While the module can get "help" from its offspring in the performance of this function, it cannot delegate this responsibility. For a given input, only the module can ensure the "delivery" of the corresponding output. A module looses control (cannot ensure correct outputs) when any of its offspring (1) stop before completion, (2) go into endless loops or, (3) do not return required information back to the module.

Axiom 3 is concerned with where the required range element (as produced by an offspring) is delivered as dictated by its module. Clearly, it is undesirable for every function in the system to obtain or alter values of every variable in the system. The ability to obtain or alter the values or variables is called access rights. According to Axiom 3, the module can assign to its offspring the right to alter the values of the output variables of the module's corresponding function (i.e. the output access rights to these variables). (The module's corresponding function, similarly, will have first secured access rights from its parent.) As a consequence of Axiom 3, each range variable (output variable) of the MCF must appear as a range variable of the function of at least one of the module's offspring.

Axiom 4 is concerned with the way that the module controls access to its domain elements (input access rights); specifically, the module can grant its offspring the right to access its domain elements for reference purposes only. The module does not have the ability to alter its domain elements. As a consequence of Axiom 4, each domain variable (input variable) of an MCF must appear as a domain variable of the function of at least one of its offspring.

As a consequence of Axioms 3 and 4, a variable cannot represent both domain and range elements.

Axiom 5 requires that the module must ensure the rejection of inputs received that are not in the domain of the MCF. A function remains undefined for elements that are outside of its domain. A module, however, in performing its corresponding function, is responsible for determining if such an element has been recieved, and, if so, it must ensure its rejection. In a sense, the improper input element *is not* in the domain of the module's intended corresponding function but *is* in the domain of the MCF.

Axiom 6 is concerned with ordering. It requires the module to control the order (which includes priority based on time, events, importance, computational needs, etc.) of invocation of its offspring and their descendants.

Table 4.2.1 summarizes the axioms of ISDS/HOS.

Table 4.2.1

Axioms of ISDS/HOS

DEFINITION: Invocation provides for the ability to perform a function.

AXIOM 1: A given module controls the invocation of the set of functions on its immediate, and only its immediate, lower level.

DEFINITION: Responsibility provides for the ability of a module to produce correct output values.

AXIOM 2: A given module controls the responsibility for elements of only its own output space.

DEFINITION: An output access right provides for the ability to locate a variable, and once located, the ability to place a value in the located variable.

AXIOM 3: A given module controls the output access rights to each set of variables whose values define the elements of the output space for each immediate, and only each immediate, lower level function.

DEFINITION: An input access right provides for the ability to locate a variable, and once located, the ability to reference the value of that variable.

AXIOM 4: A given module controls the input access rights to each set of variables whose values define the elements of the input space for each immediate, and only each immediate, lower level function.

DEFINITION: Rejection provides for the ability to recognize the improper input element in that if a given input element is not acceptable, null output is produced.

AXIOM 5: A given module controls the rejection of invalid elements of its own, and only its own, input set.

DEFINITION: Ordering provides for the ability to establish a relation in a set of functions so that any two function elements are comparable in that one of said elements precedes the other said element.

AXIOM 6: A given module controls the ordering of each tree for the immediate, and only the immediate, lower level.

4.3 Functional Decomposition

While a function can be decomposed in many ways, the HOS axioms provide rules for the construction of nodal families (i.e. the decomposition of a function). From the axioms, three primitive control structures are derived which are used for functional decomposition (HAM76b). These control structures are composition, set partition, and class partition.

Composition is illustrated in Figure 4.3.1. In order to perform $f_1(x)$, the function f_2 must first be applied to x which results in output z . z then becomes an input to f_3 which produces the desired range element of the overall function f_1 .

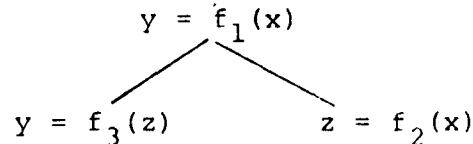


Figure 4.3.1: An Example of Composition

It is important to observe the following characteristics of composition (characteristics are explained with respect to the example in Figure 4.3.1):

- (1) One and only one offspring (specifically f_2 in this example) receives access rights to the input data, x , from module f_1 .
- (2) One and only one offspring (specifically f_3 in this example) has access rights to deliver the output data, y , for module f_1 .
- (3) All other input and output data that will be produced by offspring controlled by f_1 will reside in local variables (specifically z in this example). Local variable, z , provides communication between the offspring, f_2 and f_3 .

- (4) Every offspring is specified to be invoked once and only once in each process of performing the parent MCF.
- (5) Every local variable must exist both as an input variable for one and only one function and as an output variable for one and only one different function on the same level.

Additional examples of composition are given in Figures 4.3.2 and Figure 4.3.3.

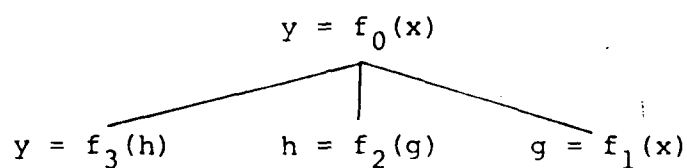


Figure 4.3.2: Composition with Three Functions on One Level

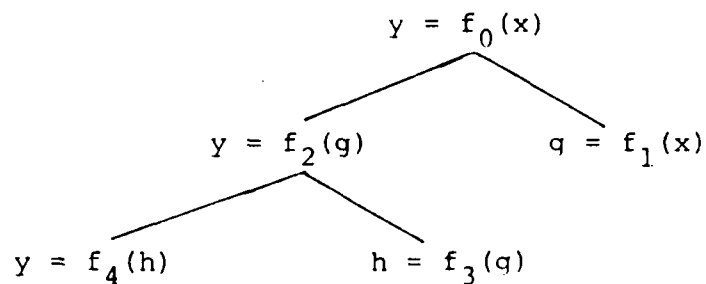


Figure 4.3.3: Multilevel Composition

Set partition, which involves partitioning of the domain, is illustrated in Figure 4.3.4. In this example, the set which comprises the domain is partitioned* into two subsets. For set partition, only one of the offspring will be invoked for each performance of the MCF at f_1 (the determination being based on the value of x received) and that offspring will produce the required range element for its parent module when it is performing.

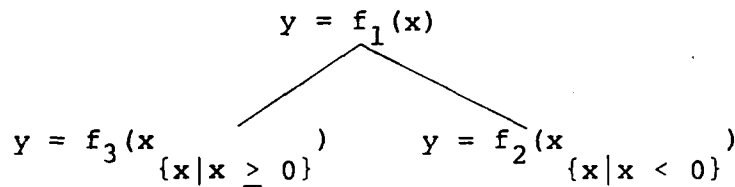


Figure 4.3.4: An Example of Set Partition

The following characteristics with respect to set partition should be observed:

- (1) Each offspring of the module at f_1 is granted permission to produce output values of y .
- (2) All offspring of the module at f_1 are granted permission to receive input values from the variable x .
- (3) Only one offspring is specified to be invoked per input value received for each process of performing its MCF i.e., only one offspring has a state change for a given state change of the parent module.
- (4) The values represented by the input variables of an offspring's function comprise a proper subset of the domain of the function of the parent module.
- (5) There is no communication between offspring.

* Partitioning implies the subdivision of the original set into non-overlapping (i.e. mutually exclusive) subsets.

Alternative approaches to the set partition illustrated in Figure 4.3.4 are presented in Figures 4.3.5 and 4.3.6.

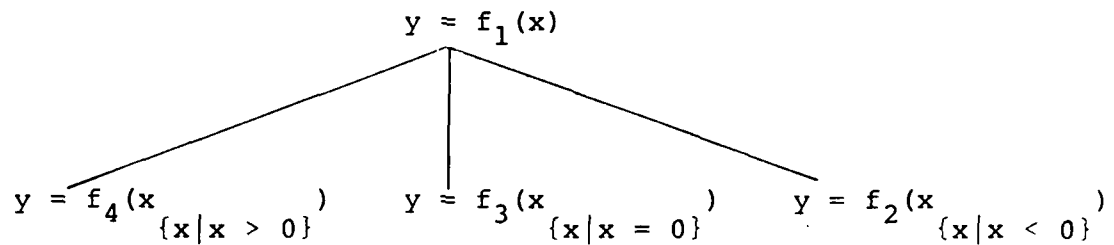


Figure 4.3.5: Set Partition with Three Functions on One Level

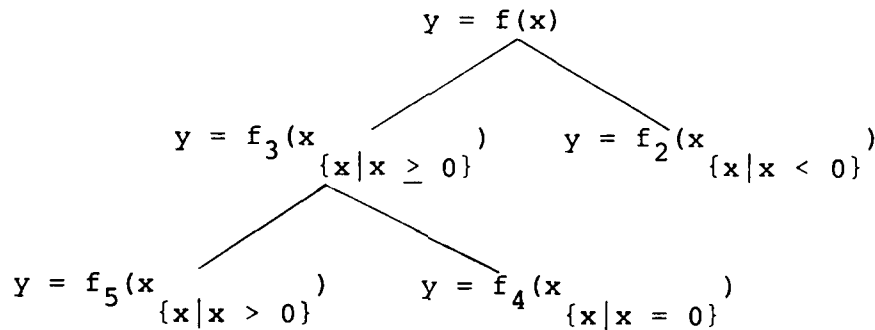


Figure 4.3.6: Multilevel Set Partition

4.4 Illustration of the Axioms

To explain the concept of control, we now illustrate the axioms of ISDS/HOS individually. Each individual axiom shows properties of control.

In these illustrations, the reader should associate the titles LT.COLONEL, MAJOR, etc., with a module and its corresponding function. For example:

PROTOTYPE = MAJOR(CONTRACT)

is analogous to

$$y = f(x).$$

People may be "allocated" to actually perform each function. We can even allocate the same person to more than one function. For the purposes of the axiom illustrations, separate the allocation concept from the function concept., i.e., LT.COLONEL is a function, not a person.

Axiom 1

Axiom 1 is illustrated with the aid of Figure 4.4.1. The function is LT.COLONEL and the input (domain) and output (range) variables are REQUIREMENT and SPECIFICATION, respectively, i.e.,

$$\text{SPECIFICATION} = \text{LT.COLONEL}(\text{REQUIREMENT})$$

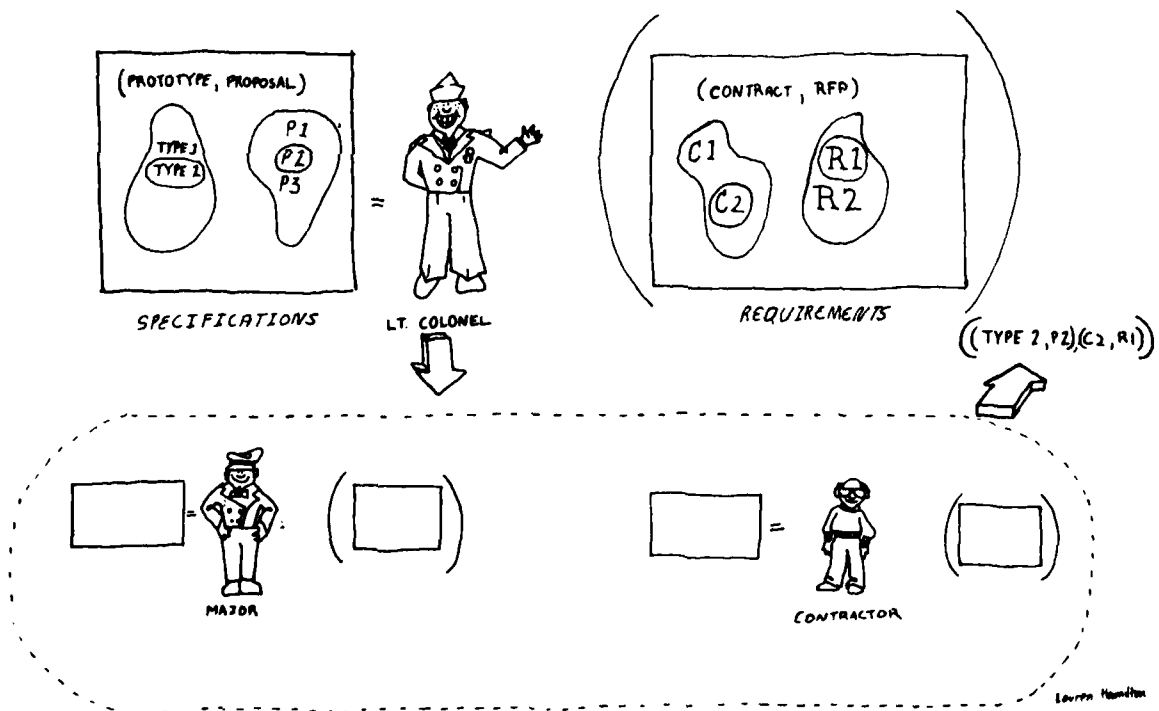


Figure 4.4.1

AXIOM ① Invocation Rights

In Figure 4.4.1 (PROTOTYPE, PROPOSAL) is a data structure* of SPECIFICATION; (CONTRACT, RFP) is a data structure of REQUIREMENT. LT.COLONEL ensures the completion of his function by getting the MAJOR and CONTRACTOR to work together to do the LT.COLONEL's job. Together, the MAJOR and CONTRACTOR use the input of the LT.COLONEL's function and produce the output for the LT.COLONEL. An instance of this effort is indicated by its ordered pair ((TYPE2, P2), (C2,R1)) pointed to by the arrow.

If the MAJOR and CONTRACTOR are the only functions that belong to a control level controlled by LT.COLONEL, then both MAJOR and CONTRACTOR contribute to completing the LT.COLONEL function. The act by LT.COLONEL of getting MAJOR and CONTRACTOR to contribute is called invocation. Axiom 1 relates each invocation to the total function of LT.COLONEL. This is illustrated in Figure 4.4.1 by the arrow pointing to the collection of functions within the dotted line. Specifically, LT.COLONEL controls only the invocation of MAJOR and CONTRACTOR and the invocation of MAJOR and CONTRACTOR is controlled only by the LT.COLONEL.

* The notion of a data structure, often referred to as an implementation of a variable, is helpful in understanding a complex set of values of a variable. PROTOTYPE and PROPOSAL are variables in their own right since a value of PROTOTYPE and a value of PROPOSAL are both necessary to complete a value of SPECIFICATION (e.g., a value of PROTOTYPE is TYPE2; a value of PROPOSAL is P2; the corresponding value of SPECIFICATION is (TYPE2, P2)).

Axiom 2

Axiom 2 is illustrated with the aid of Figure 4.4.2. The axiom requires LT.COLONEL to be responsible for relating each input value (domain element) to the correct output value (range element) (e.g. for input (C1, R1), LT.COLONEL ensures the assignment of output (TYPE1, P1)). Axiom 2 requires that, for a particular input value, there be one, and only one particular output value. LT.COLONEL must take full responsibility for the final product. He cannot delegate this responsibility to any of his subordinates or to any other module. (LT.COLONEL, however, must delegate the work. See Axiom 1.) By itself, Axiom 2 does not address the concept of control with respect to lower levels. In Figure 4.4.2, LT.COLONEL is pulling the strings; this demonstrates only LT.COLONEL's responsibility as total caretaker of the function.

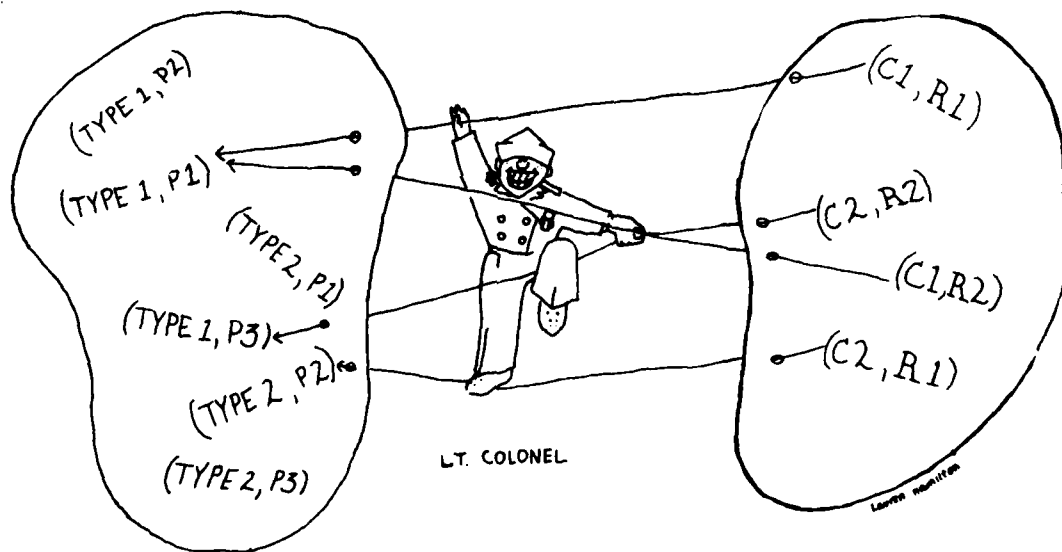


Figure 4.4.2

AXIOM ② Responsibility Rights

Axiom 3

To illustrate Axiom 3 (Figure 4.4.3), we refer to LT.COLONEL's corresponding function in terms of the data structure representation:

$$(\text{PROTOTYPE}, \text{PROPOSAL}) = \text{LT.COLONEL} (\text{CONTRACT}, \text{RFP})$$

According to Axiom 3, LT.COLONEL* grants (1) to CONTRACTOR the access rights to PROPOSAL (i.e., CONTRACTOR has the right to deliver the proposal to its assigned location), and (2) to MAJOR, the access rights to PROTOTYPE (i.e., MAJOR has the right to deliver PROTOTYPE to its intended location). Since the MAJOR has output access to PROTOTYPE, he must deliver (in a given performance) the same value of PROTOTYPE as that of LT.COLONEL's corresponding function.

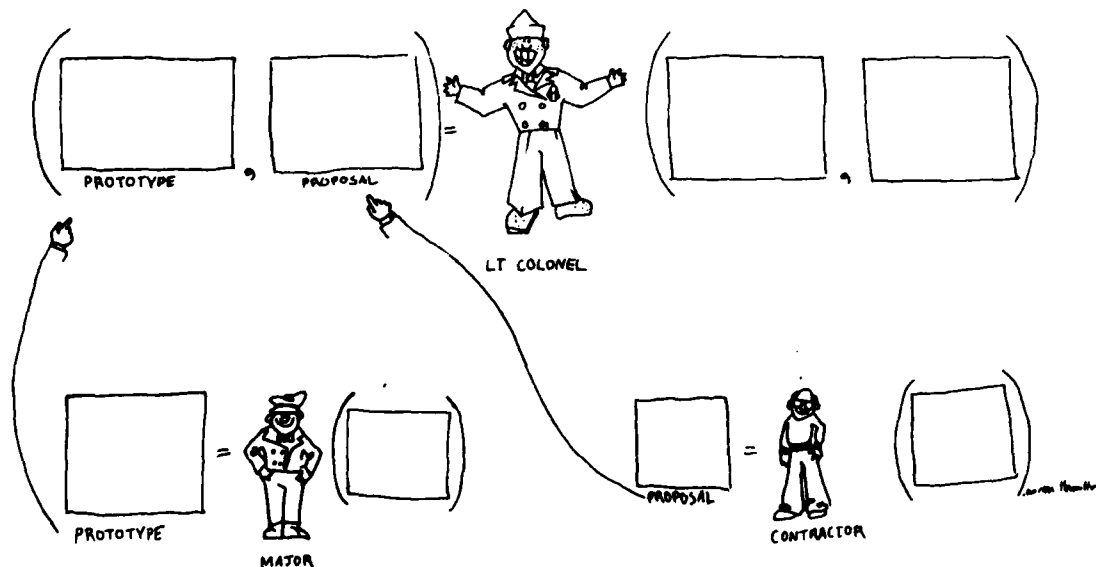


Figure 4.4.3

AXIOM ③ Output Access Rights

- * The LT. COLONEL's corresponding function will have first secured output access rights to PROTOTYPE and PROPOSAL in a similar fashion.

Axiom 4

According to Axiom 4, (Figure 4.4.4), LT.COLONEL grants (1) to CONTRACTOR the access rights to RFP (i.e., CONTRACTOR has the right to reference RFP from its assigned location) and, (2) to MAJOR, the access rights to CONTRACT (i.e. MAJOR has the right to reference CONTRACT from its assigned location). Since MAJOR has input access rights to CONTRACT, he must reference (in a given performance) the same value of CONTRACT as that of LT.COLONEL's corresponding function.

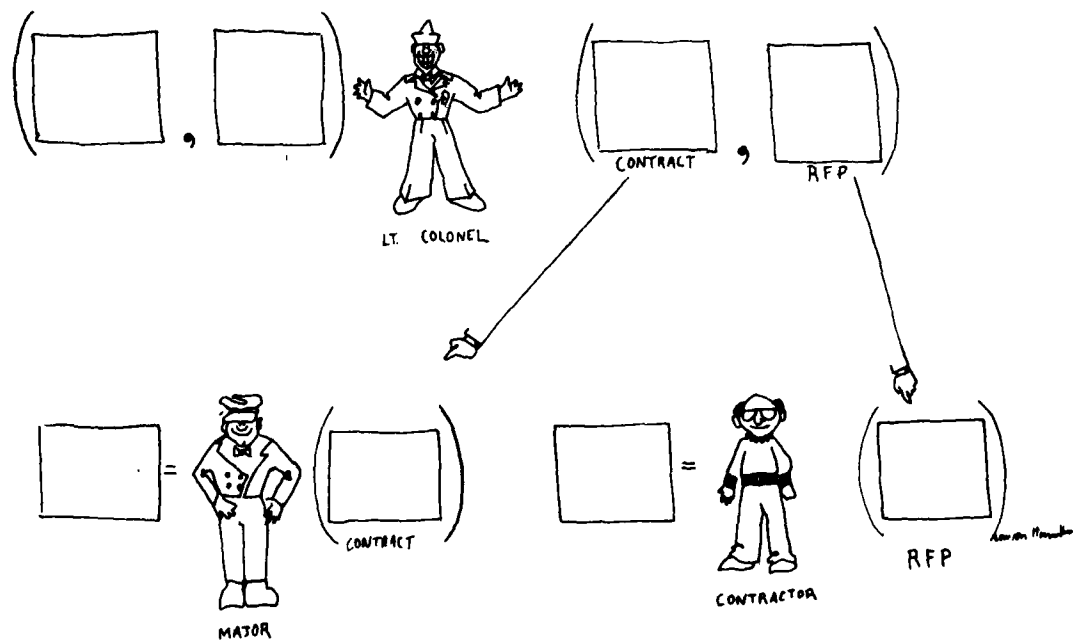


Figure 4.4.4

AXIOM ④ Input Access Rights

Axiom 5

Axiom 5 is illustrated in Figure 4.4.5. Here, we use the same function, REQUIREMENT = LT.COLONEL (SPECIFICATION), as shown in Figures 4.4.1, 4.4.2, 4.4.3, and 4.4.4.

Axiom 5 deals with the rejection of invalid input values. Techniques used to determine if input values are valid are controlled by the module itself. This means that if LT.COLONEL knows he can accept only (C1, R1), (C1, R2), (C2, R2) or (C2, R1), then he must reject anything else (see Figure 4.4.5). He must carry out this rejection without requesting any aid from his offspring, i.e., rejection control is limited to the module itself.

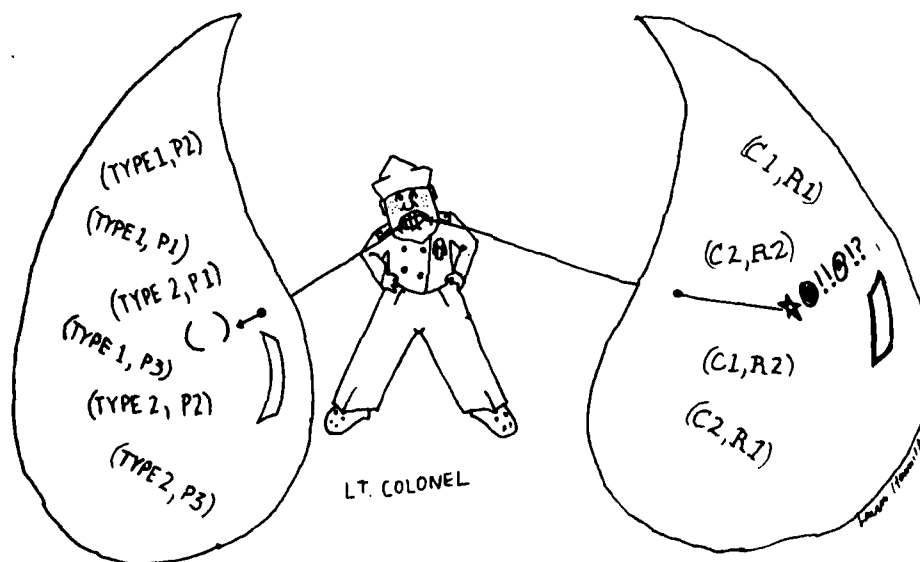


Figure 4.4.5

AXIOM ⑤ Rejection Rights

Axiom 6

Axiom 6 relates ordering of immediate subordinate offspring and their dependents with respect to the module. In Figure 4.4.6, we illustrate Axiom 6 in terms of the data structure representation of the LT.COLONEL's corresponding function.

In this example, not only are the LT.COLONEL's inputs related to his outputs; but the inputs are functionally related to each other, and the outputs are functionally related to each other.

The ENGINEER and BUSINESS.OFFICE functions in Figure 4.4.6 are external to the system and only provide an explanation of where the inputs to the LT.COLONEL's system comes from.

The inputs are functionally related to each other by means of the BUSINESS.OFFICE function:

$$\text{CONTRACT}_{\text{RFP}} = \text{BUSINESS.OFFICE (RFP)}$$

The engineer function chooses which prototype is needed for the major:

$$\begin{array}{ccc} \text{PROTOTYPE} & & = \text{ENGINEER (PROPOSAL)} \\ & \text{PROPOSAL} & \end{array}$$

The input and output values of the variables, RFP and PROPOSAL, of the CONTRACTOR function are used as input values for the BUSINESS.OFFICE and ENGINEER functions. The ENGINEER and BUSINESS.OFFICE functions produce values which become variables for the MAJOR function.

The LT.COLONEL must control his immediate subordinates so that each subordinate knows when he can start to work and the conditions under which he must complete his job. When the RFP is available from the LT.COLONEL, the CONTRACTOR can immediately start working in order to produce the proposal. If the CONTRACT has already been prepared by the BUSINESS.OFFICE when the LT.COLONEL starts

his task, the MAJOR can also begin work immediately. On the other hand, if the BUSINESS.OFFICE has a work overload, the MAJOR will have to wait longer for the BUSINESS.OFFICE to complete the CONTRACT.

Once the CONTRACT is in, the MAJOR can begin work. The LT.COLONEL insures that the CONTRACTOR must produce the PROPOSAL before the MAJOR can finish his job. For every PROPOSAL the CONTRACTOR creates, the ENGINEER uses that PROPOSAL to choose the prototype. The LT.COLONEL controls the CONTRACTOR to complete his job before the MAJOR can complete the task of building a prototype.

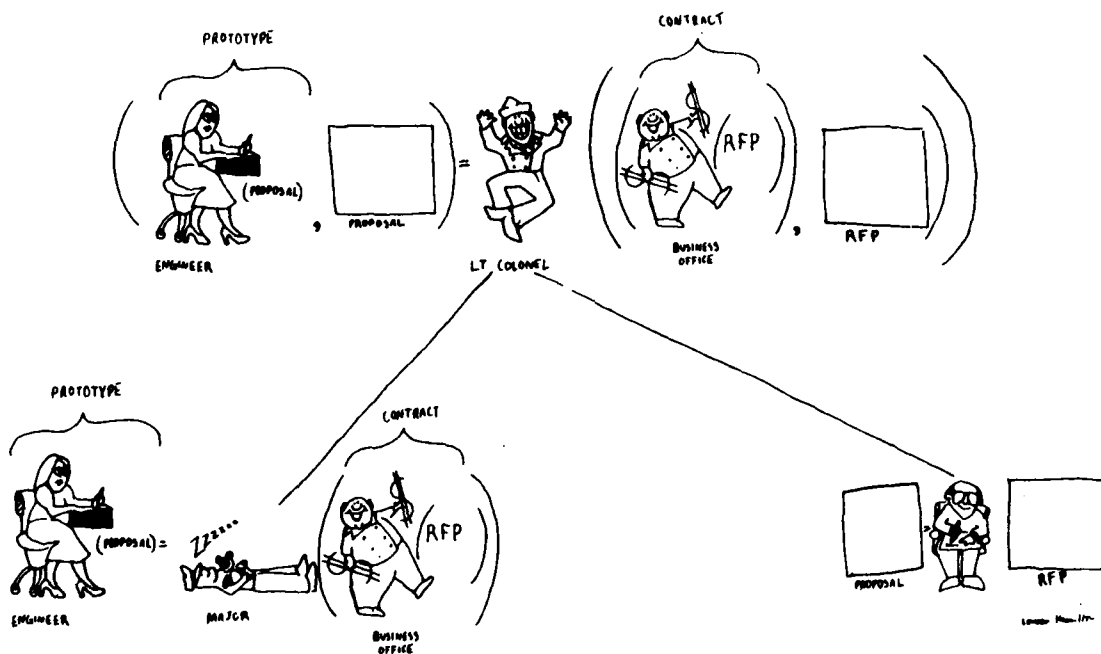


Figure 4.4.6
AXIOM ⑥ Ordering Rights

4.5 Examples

4.5.1 The BRIGGEN System

Hierarchical systems exist in many forms. Many management structures such as a military management structure, a business organization, or a government are structured hierarchically. The key to understanding the structure of any hierarchy is in determining what objects belong to the hierarchy and what relationships exist between the objects of the hierarchy.

To explain and illustrate the axioms of ISDS/HOS, we have created a fictional system called the BRIGGEN system. For the BRIGGEN system, we describe the objects which are variables, values, functions, and trees; the relationship of the hierarchy is control.

We have chosen to make the military chain of command a management structure analogous to an ISDS/HOS hierarchy. For the BRIGGEN system, the system function is to direct a research project in systems engineering to be completed one year from the start of the fiscal year. The system is decomposed into functions, each labeled to correspond to the person responsible for that function (Figure 4.5.1).

The management hierarchy shown in Figure 4.5.1 is referred to as a tree. Each member of the hierarchy (e.g., BRIGGEN, COL1, CAPT2) has a particular position of responsibility. Each member of the hierarchy controls the use of his immediate subordinates. For example, BRIGGEN controls the use of COL1, COL2, and COL3; COL1 controls the use of LTCOL1 and LTCOL2, etc. The properties of control are determined by the axioms of ISDS/HOS. When we refer to COL1 giving orders to LTCOL1 and LTCOL2, COL1 is referred to as a "controller" or as a "module". When COL1 receives orders from BRIGGEN he is referred to as a function. A module is a supervisor; a function is a subordinate.

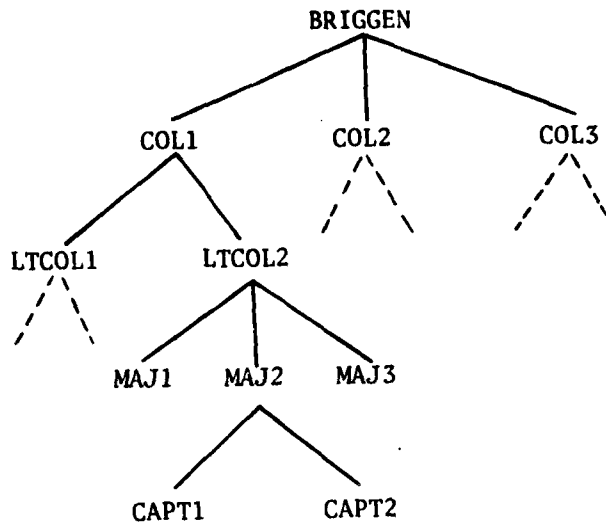


Figure 4.5.1
BRIGGEN Invocation* Tree

Figure 4.5.2 is a more detailed version of Figure 4.5.1. Figure 4.5.2 is referred to as a control map because the inputs and the outputs of the functions are provided. In Figure 4.5.2, interfaces of the functions are shown by illustrating the access rights to the variables of the functions as well as the invocation tree of the BRIGGEN management structure.

BRIGGEN has access to MEMO1, MEMO2, and MEMO3. BRIGGEN can read these memos or allow any of his subordinates to read these memos, but he cannot rewrite the memos or alter them in any way. Each memo can be delivered to BRIGGEN at different times or at the same time. If one memo arrives, BRIGGEN can allow the memo to be sent to the responsible subordinate before the other memos arrive. Likewise, if one subordinate can get his job done based on the contents of one memo, the subordinate can deliver his product to BRIGGEN as long as BRIGGEN has not imposed restrictions (e.g., timing delays) on that subordinate.

* An invocation tree is a representative control map which includes only function names.

In this project, the Brigadier General, BRIGGEN, has been asked by his superior to complete two presentations and a prototype based on the contents of three memos. BRIGGEN must complete the project one year from the start of the fiscal year.

BRIGGEN requirements include the time constraint imposed on the research project. This time constraint will be discussed in the section entitled The Initiation of Command in BRIGGEN. The structure of BRIGGEN (Figure 4.5.2) illustrates the three HOS primitive control structures: class partition, composition, and set partition (HAM76b).

CONTROL STRUCTURES OF BRIGGEN

BRIGGEN has ordered his three colonels to perform three independent functions, COL1, COL2, and COL3. BRIGGEN is using a class partition to control his subordinates. Each colonel can perform his job independent of the other two colonels. COL1 uses the information in MEMO1 to produce PRESENTATION 1; COL2 uses the information in MEMO2 to produce PRESENTATION 2; COL3 uses the information in MEMO3 to produce the PROTOTYPE. Each colonel can begin his job immediately if the BRIGGEN has all the memos available at the start of the project. If any of the three memos is delayed, the colonel responsible for the delayed memo must wait until he has the information required to begin work. BRIGGEN is responsible for informing his subordinates when to complete the job. This aspect of BRIGGEN's function is not depicted in Figure 4.5.2, but will be discussed later with respect to the Initiation of Command in BRIGGEN.

In Figure 4.5.2, we do not show how COL2 and COL3 do their jobs. Figure 4.5.2 only shows the relationship of COL2 and COL3 with respect to BRIGGEN.

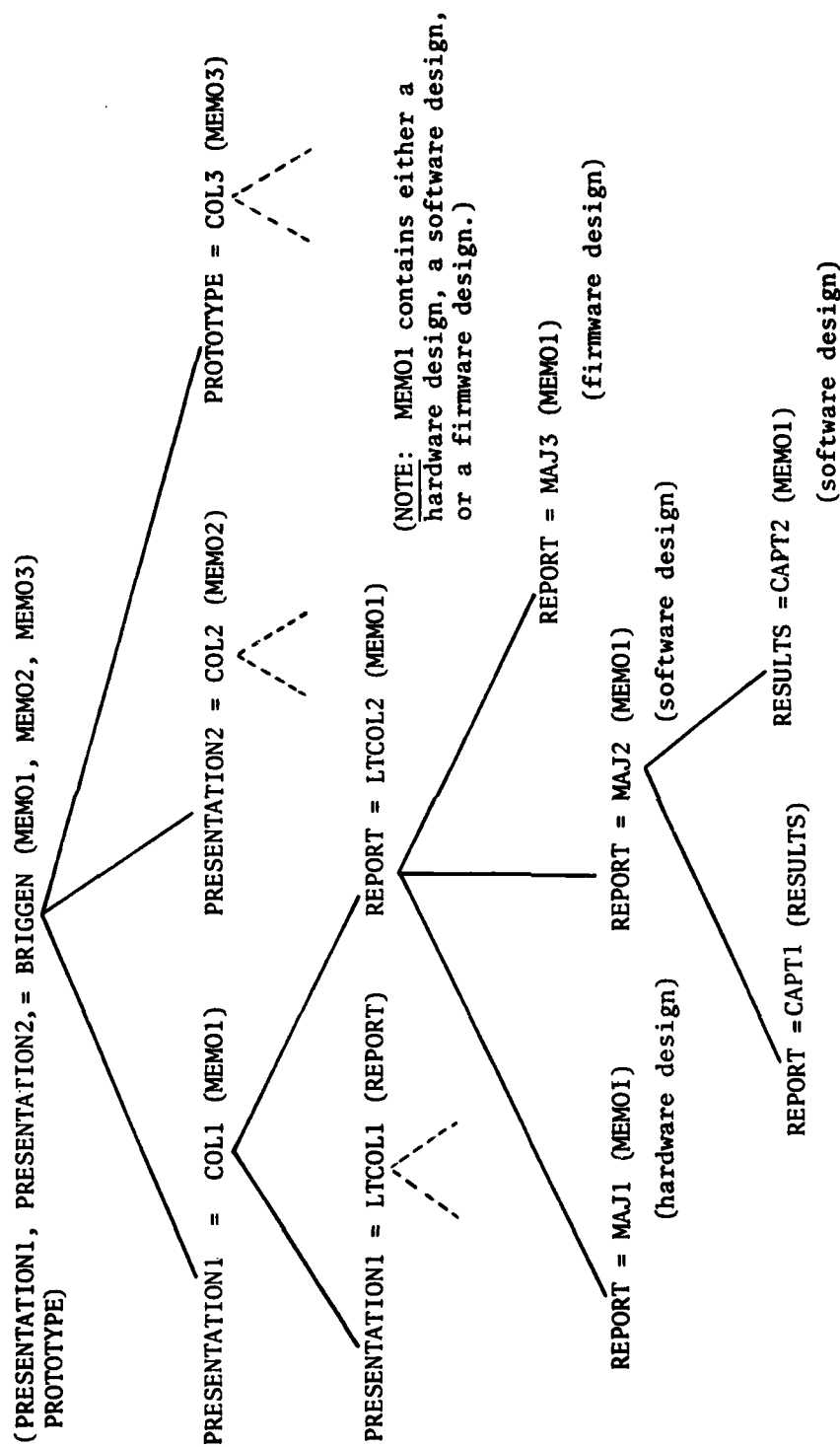


FIGURE 4.5.2: BRIGGEN Management Structure Control Map

Figure 4.5.2 does, however, indicate how COL1 performs his job. In this case, COL1 is controlling his immediate subordinates, LTCOL1 and LTCOL2, by means of a composition control structure. COL1 would like his subordinates to communicate with each other. LTCOL1 depends on LTCOL2 to get his job done. LTCOL2 receives MEMO1 from COL1 and produces an intermediate REPORT for LTCOL1's use. COL1 does not care to see the REPORT; he is interested in receiving only PRESENTATION1. The same control structures can be used for many different functions in the BRIGGEN system. For example, MAJ2 uses the same management scheme (i.e., composition control structure) as COL1 to produce the REPORT for LTCOL2.

Often, the contents of the input of a module are used to determine which subordinate should do the entire job. The management scheme used to perform a job in this manner is the set partition control structure. LTCOL2 is controlling his immediate subordinates, MAJ1, MAJ2, and MAJ3 by such a control structure. For example, the contents of MEMO1 could determine which Major must be available to do the job. Each Major has access to MEMO1, but each Major has access to the contents of MEMO1 only when the contents falls into his own sphere of responsibility. In Figure 4.5.2, MAJ1 is a hardware expert; MAJ2 is a software expert; and MAJ3 is a firmware expert. Jointly, all three Majors must account for all the information that could appear in MEMO1. When LTCOL2 finds out what kind of information is in MEMO1, he assigns either MAJ1, MAJ2, or MAJ3 to write the REPORT.

RATIONALE FOR THE MANAGEMENT STRUCTURE

To comply with Axiom 1, the colonels must use the information supplied by the Brigadier General; the Major must report it to a particular Lt. Colonel. Conversely, the Brigadier General can only ask his colonels to do his job and a colonel can request his own Majors to do his job. If a Major tried to tell any

Colonel how to do his job; attempted to give orders to another Major; or attempted to interfere with a Captain's command to his subordinates, Axiom 1 would be violated. If LTCOL2 were responsible for performance analysis, in addition to hardware, software, and firmware designs, he might not ever get his job done by one of his subordinates. If this were the case, LTCOL2 might request at least one more Major in order to comply with Axiom 1. Suppose MAJ2 assigned a third task to CAPT3 to redo the software design. In this case, since MAJ2 would be doing extra work (and possible redundant work), he would be violating Axiom 1.

Everyone in the BRIGGEN system must be a responsible person via Axiom 2. That is, there must be an output from every function in the system. Axiom 2 could be violated if, for example, a Major got sick, or a Colonel retired before his job was done, or a Lt. Colonel did not know how to do his job.

In the BRIGGEN system each controller designates the data access to his immediate subordinates and conversely his functions acquire the right to access information from their controller. Note, for example, that the input access rights of MEMO1, MEMO2, and MEMO3 can be traced down the control map (Axiom 4) and similarly that the outputs can be traced up the control map (Axiom 3).

In BRIGGEN, it is possible for more than one subordinate to have the potential to access the same input variable, as in the case of the LTCOL2 and his immediate subordinates. Here, LTCOL2 controls the access rights to the variable, MEMO1, by making sure that if one subordinate can access a particular value, no other subordinate can locate the same value. If LTCOL2 allowed MAJ2 to analyze the hardware design, LTCOL2 would get an incorrect report. This mistake would illustrate a violation of Axiom 4.

A controller can set up a communication path between his immediate subordinates (c.f. COL1 in Figure 4.5.2). That is, a subordinate, such as LTCOL1, can access an input value other than COL1's input value if the subordinate's input value has been derived from COL1's input value. The access rights to the variable used to communicate the derived value are controlled by COL1 in that such a variable is never used to locate COL1's output; and such a variable can be used to communicate between only two of his immediate subordinates.

Axioms 3 and 4 prevent conflicts in the use of data resources. Suppose that LTCOL1 had PRESENTATION1 typed on the same paper as the report had been typed on. As parts of PRESENTATION1 are prepared, relevant sections of the REPORT or PRESENTATION1 would be hard to decipher correctly. Axiom 4 distinguishes REPORT as input to LTCOL1 only. The value in REPORT cannot be changed after LTCOL2 produces his output. Axiom 3 requires PRESENTATION1 to be different from REPORT in that it is only produced by LTCOL1. Although such an example appears extreme in a management scheme such as BRIGGEN (we assume a plentiful supply of paper), this example has characteristics typical of a software system resource allocation problem.

If BRIGGEN were to be modified as in the following examples, Axioms 3 and 4 would be violated.

- 1) Suppose COL3 were to read MEMO4 as well as MEMO3. BRIGGEN has no control over the access of MEMO4. In this case, BRIGGEN would not know the implications of MEMO4 to BRIGGEN's job. If COL3 uses MEMO4 to do his job, COL3 could devise many different prototypes for BRIGGEN depending on the contents of MEMO4, as well as the contents of MEMO3 (i.e, BRIGGEN could get more than one prototype for the same value of his input). BRIGGEN could not control the contents of the prototype. Most likely, irrelevant detail or artificial constraints would be introduced by COL3 which would jeopardize the quality of the research project.

- 2) Suppose COL1 were to be able to read MEMO2 instead of MEMO1. Here, BRIGGEN would not be controlling access to MEMO1. Since no one would be able to use MEMO1, the research project would essentially ignore that information. Real constraints to the project would be neglected which might cause BRIGGEN to oversimplify the significance of the research. In such a situation, BRIGGEN might also cause unnecessary arguments between COL1 and COL2 because, both being conscientious, they would both want to begin using MEMO2 at once. If BRIGGEN were to keep his subordinates happy, he would have copies made of MEMO2 so that both COL1 and COL2 could use the contents of MEMO2 whenever they wished.
- 3) Suppose MAJ2 were to present his product as a BRIEFING instead of a REPORT. Since LTCOL2 would be counting on MAJ2 to produce the REPORT if MEMO1 contained a software design, LTCOL2 could not perform his job if MAJ2 were able to decide to produce a BRIEFING instead of a REPORT.
- 4) If LTCOL1 and LTCOL2 worked together to produce the REPORT, how would PRESENTATION1 be produced? One of COL1's subordinates must prepare his presentation in order for COL1 to do his job.
- 5) Suppose BRIGGEN were to have input access rights to RESULTS in addition to MEMO1, MEMO2, and MEMO3. If the functions of COL1, LTCOL2 and MAJ2 were not modified to be able to access RESULTS, then CAPT2 would have to do the job of obtaining RESULTS over again. Not only might CAPT2 be doing an unnecessary job, but since he has access to less information than BRIGGEN, he might not get the same RESULTS that BRIGGEN started with; and there would be no means to check the validity of RESULTS.

- 6) Suppose BRIGGEN had been asked by his superior to produce REPORT in addition to PRESENTATION1, PRESENTATION2 and PROTOTYPE. If COL1 were unaware of this modification to the project, he would keep the REPORT in his files and not make a copy of the REPORT for BRIGGEN. If BRIGGEN asked COL2 for a copy of REPORT, COL2 would be at a loss as to how to respond to such a request because COL2 never communicates with COL1.

It is not always possible for a subordinate in the BRIGGEN system to produce a good result. In this system, a subordinate is responsible for recognizing the validity of his input and for taking action if he does not receive the proper information (Axiom 5). Suppose MEMO1 were to contain a set of 'humanware' values. LTCOL2 would have to recognize that he cannot do his job with such information. The LTCOL2 takes action and sends a REPORT informing LTCOL1 that he cannot supply LTCOL1 with a comprehensive REPORT. LTCOL1 is responsible for sending a message to COL1 that PRESENTATION1 will not get done. Likewise, COL1 sends a message to BRIGGEN indicating that PRESENTATION1 cannot be completed properly, making for a very unhappy Brigadier General.

A more sophisticated system would provide recovery from this problem. COL1 might be modified to accept 'performance analysis' values and call on another subordinate, such as LTCOL3, to prepare PRESENTATION1 under such conditions. In an even more sophisticated system, there might be deadlines to adhere to which could result in many more alternate ways to recover from such a situation. In the BRIGGEN system, LTCOL1 could not be permitted to check the contents of MEMO1 for validity, because LTCOL1 would be meddling in LTCOLs's domain.

In every system there must be order. Each manager must know the conditions under which he can expect each result produced by the system functions. Some orderings of output are more apparent than others in the above examples. For example, it is clear that LTCOL2 must deliver his report to LTCOL1 before LTCOL1 can prepare his presentation. Other cases are not so apparent. For example, two presentations are to be made to the Brigadier General. If they both occurred on the same date at the same time the Brigadier General could not attend both presentations. But then again, if COL1 and COL2 began speaking at once, the Brigadier General would be quite confused. When a presentation and a prototype occur simultaneously, it is not always obvious as to why they should occur in any given order. However, it might make more impact if one occurred before the other. Also, there may be limited resources available to the Brigadier General. If, for example, the functions of COL1 and COL3 were performed by one person, that particular person must know the priority of each function assigned to him so that he knows how to use his time properly. If there were no way to determine the ordering of functions in the system, or if they were ordered improperly, the manager in charge would be violating Axiom 6. If BRIGGEN had less authority than COL3, BRIGGEN could not tell COL3 when to produce the PROTOTYPE. If BRIGGEN had less authority than CAPT1, then CAPT1 could tell the Brigadier General when to do his job. Such inconsistencies in the chain of command would only occur if Axiom 6 were to be violated.

THE INITIATION OF COMMAND IN BRIGGEN

Thus far, we have discussed the BRIGGEN system with respect to several hierarchical levels of control, but we have not discussed that system with respect to the time constraint imposed on the research project. Let us consider the time requirements of the BRIGGEN system (Figure 4.3.3).

Here, the MAILMAN delivers BRIGGEN's INPUT (i.e., MEMO1, MEMO2, and MEMO3) at time, TIME_S. To distinguish this type of input/output relationship from others, we use a subscript attached to the output variable. Thus we rewrite the above equation as

$$\text{INPUT}_{\text{TIME_S}} \approx \text{MAILMAN}(\text{TIME_S})$$

In Figure 4.5.3 the MAILMAN function is not shown, but implied via the use of the subscript. BRIGGEN has been told that on the day the project is to start, he will receive the information he needs by mail. According to Figure 4.5.3, BRIGGEN will start his job when TIME_S has the value October 1, 1980. Although BRIGGEN could receive the three memos at different times if the MAJGEN wished, in this case, BRIGGEN has been restricted to receive all three memos on October 1, 1980.

BRIGGEN OUTPUT is related to the completion time of the project, TIME_C:

$$\text{OUTPUT}_{\text{TIME_C}} = \text{MESSENGER}(\text{TIME_C})$$

When the project is completed, the MESSENGER delivers the OUTPUT for BRIGGEN.

The CLOCK function is used by the Major General to initiate BRIGGEN and control BRIGGEN's completion time. We call a function such as CLOCK an effector function with respect to BRIGGEN. The BRIGGEN function is referred to as an affector function (because BRIGGEN's input/output relationship, although independent of CLOCK with respect to the values of INPUT and OUTPUT, is dependent on CLOCK in order to "run") with respect to CLOCK.

RESOURCE ALLOCATION

We have shown in system BRIGGEN how to set up the functions that need to be completed and how these functions are initiated. To actually do the job, resources such as time and space must be assigned for each function. Time, in the case of a management structure is usually addressed in terms of man hours. In the case of computers, time is allocated in terms of CPU. In a management structure, we need enough offices, desks, etc. for space. In the case of computers, we need enough computer memory.

Suppose COL3 does not receive MEMO3 until February 1, 1981. Then COL3 would have eight months to complete his job. In addition, suppose COL3 had originally intended LTCOL3 and LTCOL4 to do COL3's job. LTCOL3 needed ten months to do the work required of Section 1 and Section 2 of MEMO3 and LTCOL4 needed seven months to do the work required of Section 3 of MEMO3. In order for COL3 to adjust to his new requirement, LTCOL3 would either be over-worked or else he could not complete his job on time. One solution would be that COL3 ask for more manpower and ask LTCOL3 to concentrate on Section 1 while a new person, LTCOL5 concentrated on Section 2. For this solution, we have restructured by adding additional manpower.

In the implementation of an ISDS/HOS system, it is possible to share common resources.

For example, suppose MEMO1 were stored in a file cabinet while CAPT2 was using that information. From observing the system structure we can determine that once CAPT2 produced the RESULTS, MEMO1 would no longer be needed and, therefore, does not have to be saved in the active file cabinet. We can now make room for REPORT to be stored in the same place in the file cabinet that had once been reserved for MEMO1.

In an ISDS/HOS software system we can make use of the system structure properties in a similar manner when we allocate memory resources.

Using the BRIGGEN structure we can also allocate time resources efficiently by investigating system properties. For example, COL1, COL2, and COL3 could all work at the same time if three different people were associated with the three functions, COL1, COL2, and COL3 respectively. Suppose on the other hand, BRIGGEN had only two people to allocate to his functions. In addition, suppose function COL1 needed six months to do his job; function COL2 needed five months to his job; and function COL3 needed eleven months to do his job. Since the research project must be completed within one year, BRIGGEN could assign one person to functions COL1 and COL2 and the other person to function COL3. In this way, BRIGGEN could fulfill his requirements.

In a software system, ISDS/HOS tools determine the best way to use time resources in a similar way.

In the implementation of an ISDS/HOS system, it is also possible to share common operators. For example, in the BRIGGEN system, both LTCOL1 and LTCOL2 could use the same typewriter to type

the report or the presentation. We could determine this because the structure shows that the report must be completed before the preparation of the presentation is begun.

CHAIN OF COMMAND

Although an ISDS/HOS system requires each controller to "go through a chain of command" to carry out his necessary functions, it is not necessary for the General to actually talk to a Colonel and a Colonel to talk to a Lt. Colonel, etc. in order for a Captain to obtain his orders. If the General wanted Captains to receive a memo, the Captains could receive the memo, or copies of the memo directly, as long as access rights had been established for the Captains throughout the entire chain of command. If the General wanted to send an order via the loud speaker to everyone involved in the research project, he could do so by effectively setting up inputs to everyone in the project where each person's unique input is a unique function of the same originating loud speaker input. In this way, the BRIGGEN system could be compared to a system of several processors receiving information that is in a practical sense received simultaneously.

In an ISDS/HOS system, it is absolutely clear just who the boss is, and the communication lines in a system are clearly defined. In system BRIGGEN, each subordinate always has the same boss. Thus, subordinates have no need to question where the orders come from since they always come from the same place. Every person in the system always communicates via his own communication channels. Not only must the communication path be appropriate, but the messages transmitted on these channels must be germane.

BRIGGEN SOFTWARE SYSTEM

BRIGGEN has been presented as an ISDS/HOS military management system. If we were to model the functions of this system on a

computer, the BRIGGEN functions in the software system would be the same functions as those in Figure 4.2.1. The software system that represented BRIGGEN would have the same characteristics as the BRIGGEN management system.

4.5.2 The Line Justifier (GRI76)

The Line Justifier system is to insert blanks between words of a line so that the last character of the last word appears in the last column of the line. In addition, the following constraints are imposed: (1) the number of blanks between different pairs of words on a line may differ by no more than one and (2) for odd (even) lines more blanks are inserted toward the right (left) of the line.

Assume the above paragraph (taken from GRI76) stated the initial requirements of a problem. After manually examining sentences of a random chosen paragraph, these requirements were found to be incomplete. Therefore, the following additions to the requirements were made: (1) the first word always remains where it started, (2) only one blank separated words on the original unjustified line* and (3) the punctuation marks , , , , ? , ! , ; , : , " are part of the word on its left while " is part of the word on its right. In addition, it was ascertained that what was provided to the line justifier was (1) the column numbers where the words begin, (2) the total number of columns and (3) the length of the last word on the line.

*For the purpose of this discussion, this requirement was added to simplify the problem.

In what follows, the line-justifier will be explained by walking through the process of building its tree.

The first step is to represent the system as the root of the tree in which the domain and range is specified (Figure 4.5.2.1). In addition, a supporting narrative is developed (i.e. the key of Figure 4.5.2.1).

$A_N = \text{LINE_JUSTIFIER}(L, S, C_N)$	
<u>Key</u>	
L	= length of the last word
S	= total number of columns on the line
N	= number of words on the line
C_N	= an N-tuple of variables whose values represent the column numbers where the words begin on the unjustified line
A_N	= an N-tuple of variables whose values represent column numbers where the words begin on the justified line

Figure 4.5.2.1: The Initial Assumption of Line Justifier together with its Supporting Narrative

The next step is to use one of the primitive control structures to decompose the system (Figure 4.5.2.2).

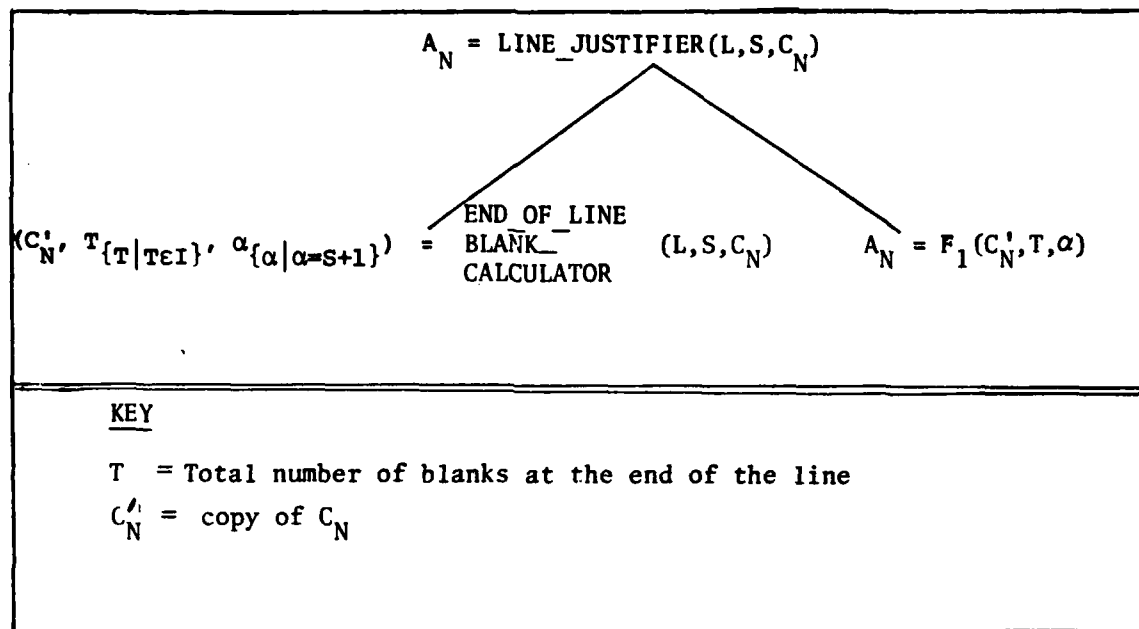


FIGURE 4.5.2.2: Line Justifier Decomposed Using the Composition Primitive Control Structure

The above decomposition was determined by the realization that a local variable T (the total number of blanks occurring after the last word) was needed in order to determine the type of processing required (eq. if $T = 0$, then the problem is solved and hence no further processing is required). The function `END_OF_LINE_BLANK_CALCULATOR` produces the value for the local variable, T , which is then communicated to the function F_1 . This function, F_1 , completes the requirements for its MCF.

We continue to apply the primitive control structures and decompose F_1 as shown in Figure 4.5.2.3.

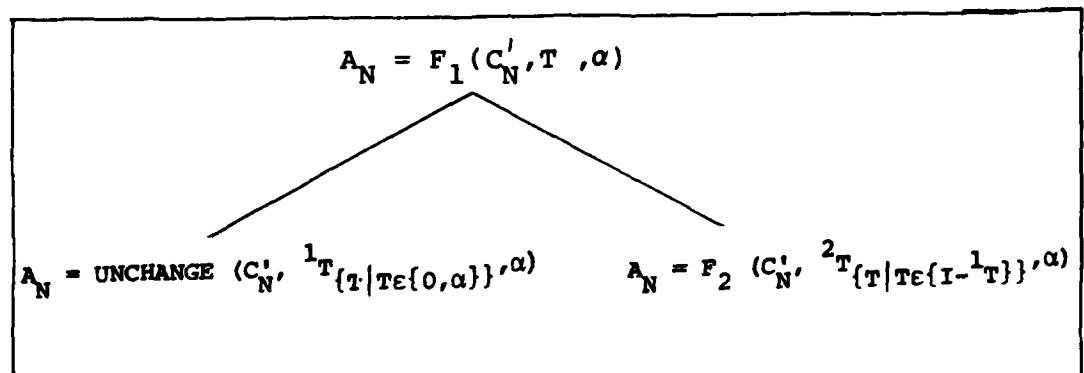


FIGURE 4.5.2.3: F_1 Decomposed Using the Set Partition Primitive Control Structure

Set partition on variable T was used to decompose F_1 (i.e., 1T and 2T are non-overlapping subsets of T ; ${}^1T \cup {}^2T = T$) so that the module F_1 , which has the responsibility to insure that its corresponding function is carried out, can decide whether the original line was already in the required format (in which case it would have invoked function, UNCHANGE) or that additional processing is required (in which case it would have invoked function F_2).

Another set partition must take place to account for the possible condition that only one word happens to occur on the line given. This specification illuminated an inconsistency in the original requirements. If the line contains only one word, then the last character of the last word cannot possibly appear in the last column of the line if at the same time the first word always remains where it started. (Except in the unlikely case where the length of the word equals the total length of the line.) If only one word does occur in the line, then a design choice must be made between these conflicting requirements. In this case it was decided that the line should be returned unchanged according to the decomposition of F_2 in Figure 4.5.2.4.

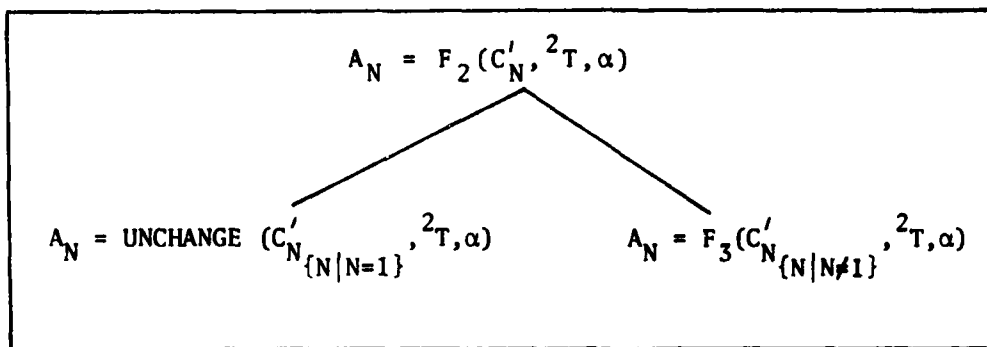


Figure 4.5.2.4:
Set Partition Decomposition of F_2 to Return Lines
Containing Only One Word

F_3 is now decomposed (Figure 4.5.2.5) by the realization that two local variables would be needed for determining A_N . These variables are (1) b , the number of blanks to be inserted between each word and (2) r , the number of word pairs requiring one additional blank between them after the inserting of the b blanks. These local variables were needed in order to fulfill the requirement that the number of blanks between different pairs of words on a line may differ by no more than one. Note that F_3 is decomposed by using the composition primitive control structure.

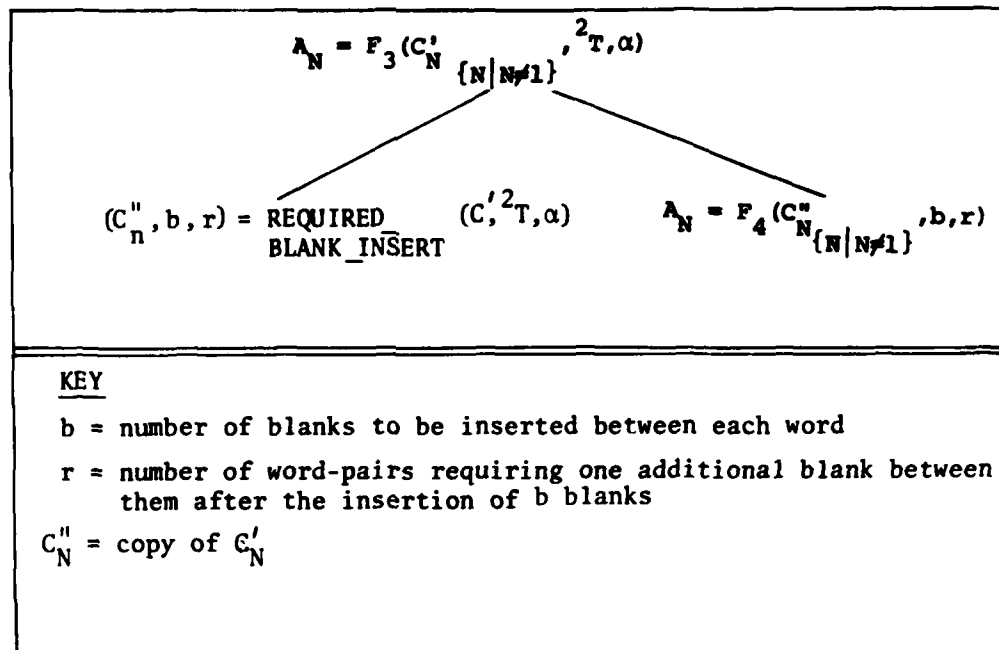


FIGURE 4.5.2.5: Decomposition Using the Composition Primitive Control Structure

At this point, an attempt was made to insert the required blanks between the words in order to meet the requirements. However, the requirement that for odd (even) lines, more blanks had to be inserted toward the right (left), of the line, could not be determined with the previously defined input. An additional input, P, was required. P represents the parity of the line indicating whether it is even or odd. It was then necessary to iterate the design process by adding the variable P to line-justifier's domain variables (Figure 4.5.2.6).

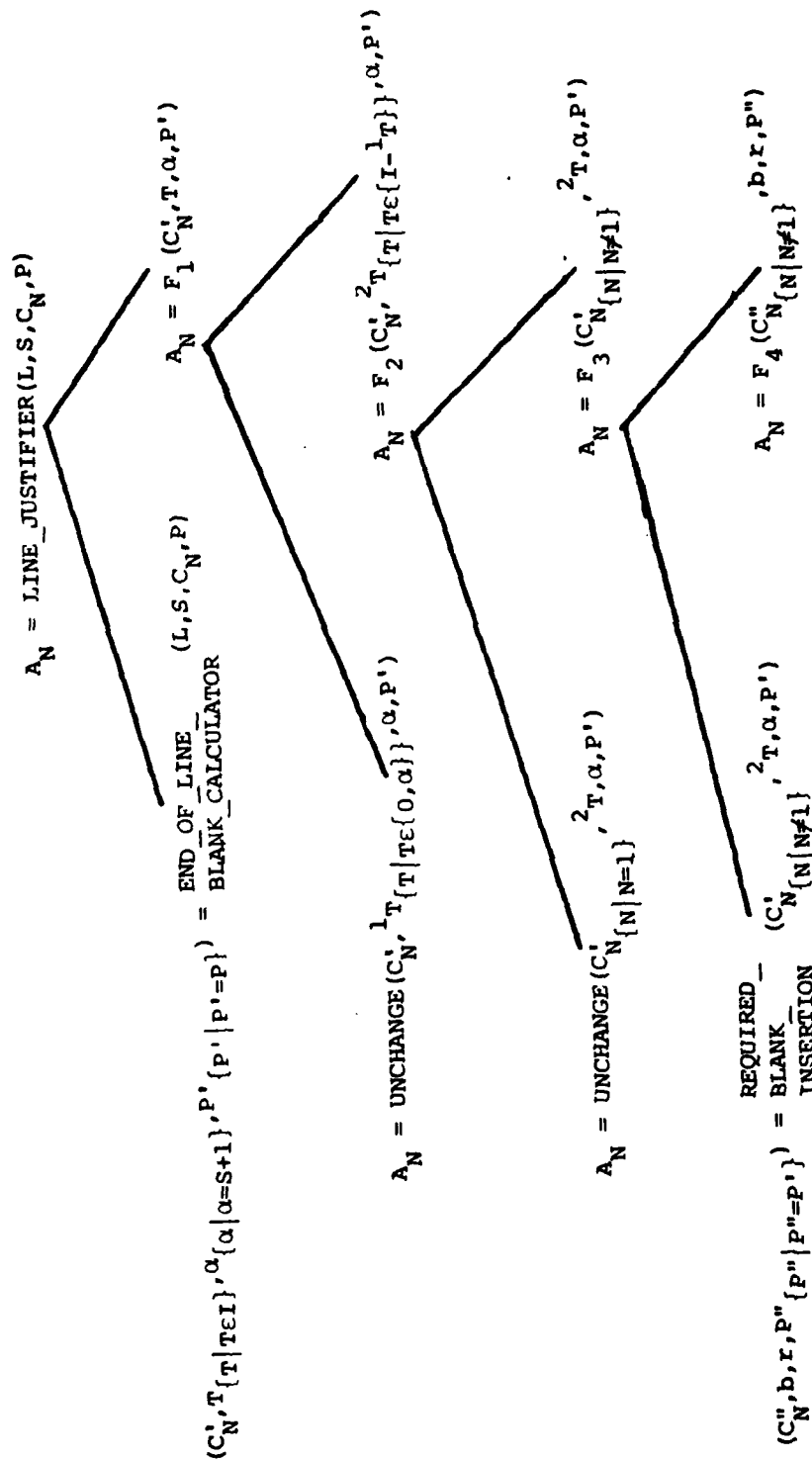


Figure 4.5.2.6: Restructuring of the Control Map to Include Line Parity Requirement

Now we explicitly demonstrate that last requirement by decomposing F_4 using P depending whether the lines parity is odd or even as shown in Figure 4.5.2.7.

At this point, it was felt that the requirements for the line-justifier were completely and explicitly stated. Therefore, the decomposition process for stating the line-justifier's functional requirements was stopped. The complete tree in the form of an invocation map is shown in Figure 5.4.2.8.

Using the system control map as a guide to design, the HOS code was produced as seen in Figure 4.5.2.9.

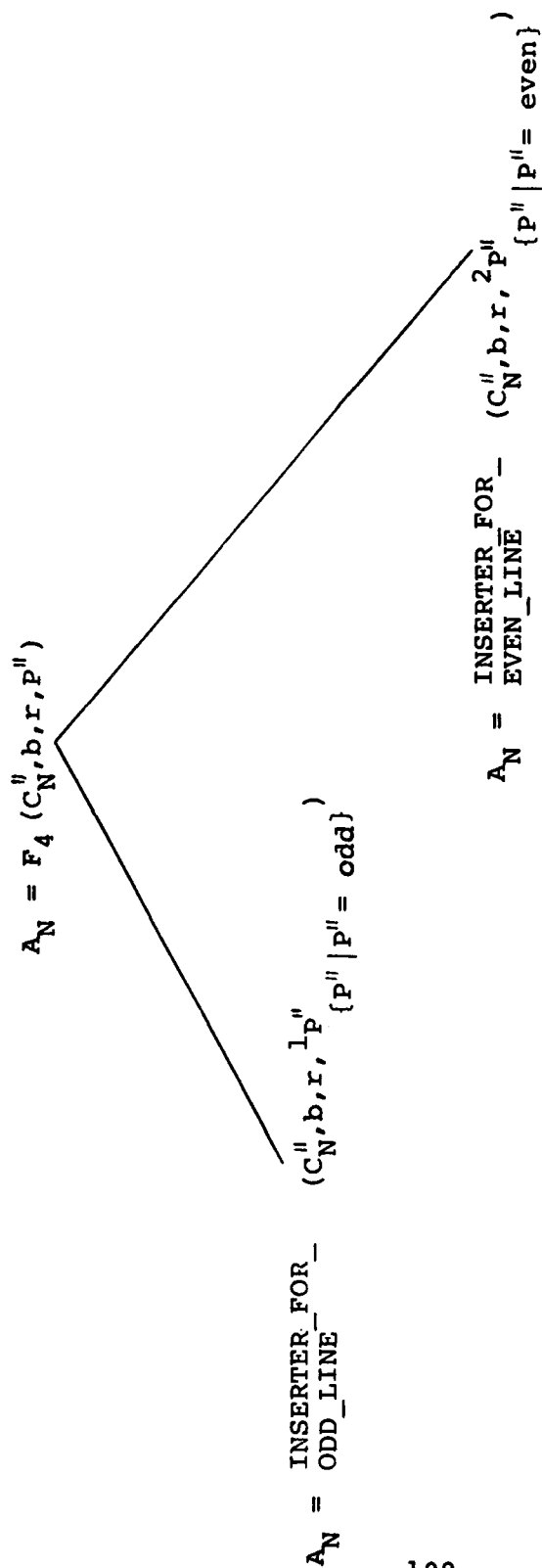


Figure 4.5.2.7: Line Parity Set Partition of F_4

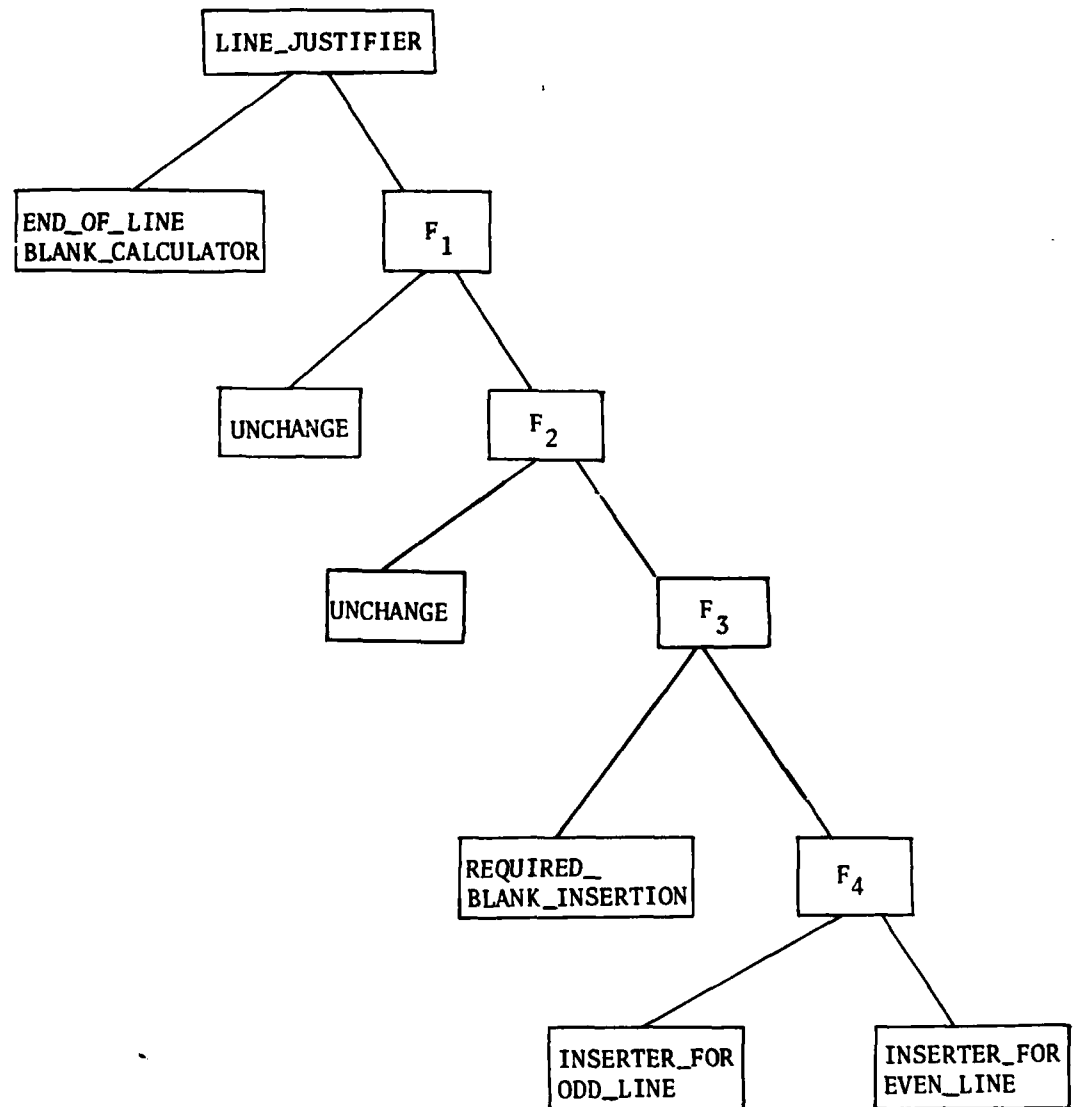


Figure 4.5.2.8: Invocation Map for Line Justifier

```

LINE_JUSTIFIER: PROCEDURE(C,L,S) ASSIGN(C);
DECLARE A,C ARRAY(*);
DECLARE B,L,N,S,T INTEGER;
  N = SIZE(C);
  T = S-(C(N)+L-1);          /* no. of blanks left at end of line */
  IF T = (S+1) | T=0 | N=1   /* line was empty or words were justi-
                              fied or only one word was on line
                              (which may be all blank) */
  THEN RETURN C;
  ELSE
    DO;
    B = T/(N-1);
    R = REMAINDER(T/N-1);
    DO I=2 TO N;              /* insert b blank between each word */
      A(I) = C(I)+(I-1) B;
    END;
    IF LINE = ODD
    THEN
      DO;
      DO I = 1 TO R;
        A(N-R+1) = C(N-R+1)+I;
      END;
      RETURN A;
      END;
    ELSE                      /* line is even */
      DO;
      DO I=1 TO R;
        A(I+1) = C(I+1)+I;
      END;
      DO I = R+2 TO N;
        A(I) = C(I)+R;
      END;
      RETURN A;
      END;
    END;
  END;
ALL_DONE:      CLOSE;

```

Figure 4.5.2.9: HOL Code for Line-Justifier System

5.0 THE USE OF ISDS/HOS DURING THE LIFE-CYCLE
OF COMPUTER-BASED MILITARY SYSTEMS

5.0 THE USE OF ISDS/HOS DURING THE LIFE-CYCLE OF COMPUTER-BASED MILITARY SYSTEMS

Despite the high level of sophistication of contemporary systems analysis, the field suffers from a serious defect. The system-specification process is itself a system, but, ironically, it is generally carried out in an unsystematic fashion.

Much of what systems designers could learn from each other is often lost in the shuffle; new systems must commonly be started from scratch. There has been no way to guarantee the efficiency of a system ahead of time. There have been problems of interface correctness, especially in complex systems designed by a large group of individuals, and there can be subsystems included which are superfluous. Overspecification of a software system can detract from its transferability from one machine to another. The failure to separate specification clearly from implementation thus can unintentionally rule out the most efficient implementation of a given system.

Let us say that a system specification is functionally adequate, if it does what its designer wanted it to do, that is, if it does carry out the function it was supposed to. There is little doubt that most systems in use today are functionally adequate, in this sense. Otherwise, they would not be in use at all. Let us also say that a system specification is fully adequate, if it does what it is supposed to do in the most effective and efficient possible way. As noted in the last paragraph, though functionally adequate, most software systems in use today most likely are not fully adequate. For all the reasons noted and others, although the jobs software systems are intended to do get done, they get done with a lot of waste of time, money and manpower.

The purpose of developing a standardized system-specification methodology is to eliminate this waste. Given some generally applicable principles governing the specification of systems,

we can reduce the problem of guaranteeing full adequacy to that of guaranteeing functional adequacy. With the correct set of principles on possible (allowed) system specifications, we can guarantee ahead of time that any system defined in accordance with those principles that does what it is supposed to do automatically does so in the most effective and efficient possible way.

We can get a clearer idea of what a systems methodology would be by considering explicitly what it is not. A priori one might interpret the term "methodology" in either of two possible ways. The most ambitious form of methodology one might hope to develop would be a discovery procedure* (CH057) for system specifications. A discovery procedure would be a mechanical (algorithmic) procedure or set of procedures that would automatically produce, from a given set of requirements and specifications, a system that would produce those specifications from those requirements. Ideally, if we could manage to develop such a discovery procedure, we could eliminate systems analysts and designers altogether. The discovery procedure would automatically provide the appropriate system for any desired purpose. At our present level of knowledge, however, and probably in principle, such a notion of methodology is unrealizable. The most we can hope for at this time is a theory of constraints on system specifications. Such a theory would severely limit the kinds of system specifications an analyst could design. If the system specification is functionally adequate, and if the designer has adhered strictly to the constraints provided by the theory, then the theory would guarantee that it is fully adequate as well.

Developing such a theory of constraints would place systems analysis on a par with the already developed natural sciences. When a physicist or chemist performs an experiment and observes a new

* It is worth noting that the tremendous growth and development of linguistics that began in the late 1950's was a direct result of the explicit rejection of the discovery-procedure notion of methodology in favor of that of a theory of constraints.

phenomenon, for example, s/he tries to construct a theory that explains it. There is no discovery procedure that automatically produces a theory from the observations. The human scientist must use his/her ingenuity to construct the theory, just as the human systems analyst must use ingenuity in designing a system. What the scientist does have available, however, is a theory of constraints on possible theories that limits the options available. Any theory the scientist proposes must guarantee conservation of mass-energy and of momentum, for example, and must be consistent with the laws of thermodynamics. These principles serve as axioms, so to speak, which any acceptable scientific theory must satisfy. What we need in systems analysis, analogously, is a set of axioms (principles) which any fully adequate system specification must satisfy.

We are using the term "methodology," then, in exactly the sense in which it is used in the natural sciences. Which specific principles we will have to recognize as the axioms of our methodology is, as in any beginning science, an empirical question.

In order to automate the process of developing systems, we need a methodology for defining systems which are understood by automated tools, i.e., ISDS/HOS. The methodology of ISDS/HOS is used throughout all phases of a given system development. With the methodology of ISDS/HOS, we apply the same axioms and therefore the same decomposition techniques of ISDS/HOS throughout an integrated system development to define:

the target system - the system whose functions define the mission (i.e., the application which is the focal point of the development process);

real-time support systems - the support systems (such as an operating system (OS) or interpreter) which are resident in the target machine when the target system is deployed;

non-real-time support systems - the support systems (such as a compiler or a static analyzer) which are not resident in the target machine when the target system is deployed. These systems might

be able to be run on the same machine as the target system in non-real time or exist in a host computer. These support systems are used only to develop target applications;

the personnel management system - the system which reflects the management structure of the personnel who direct all of the development efforts of the target system and all the development efforts of the systems involved in supporting the target systems;

the environment system - the system which represents the environment within which the target system resides;

the development process - this is a system whose functions define the process for each phase of development from the conceptual phase to the final delivery and maintenance of the target system;

the building process - the system which defines the functions of building libraries of the target system and target support systems modules, and building assemblies of subsystems to form an official assembly of 'Frozen' modules;

source systems - the systems which produce the requirements for the target system;

the disciplines of development - the subsystems of the development process which define the process steps within each phase of development; these disciplines are design, implementation, verification, management and documentation.

combinations of the above systems - various systems and subsystems can be combined to form new systems, (e.g., an environment system can be combined with a target system to form a simulated system of a target system in its real environment);

the definition of the development process - this is a system which defines the methodology of putting together systems (e.g., the definition of ISDS/HOS).

With the methodology of ISDS/HOS, we use the same tools and techniques to define and describe functions and interfaces of a system throughout all phases of a system development.

With the methodology of ISDS/HOS, we use the same tools and techniques to implement and describe the execution flow of a system throughout all phases of a system development.

With the methodology of ISDS/HOS, we use the same tools and techniques to verify and describe the verification processes for a system throughout all phases of a system development.

With the methodology of ISDS/HOS, we use the same tools and techniques to manage and describe the management processes for a system throughout all phases of a system development.

Before we discuss the development properties of ISDS/HOS, we will first discuss the problems involved in developing a system. In what follows, we will discuss what we mean by a system and by the environment within which a system resides; we will also discuss the types of requirements that are involved in developing a system. These requirements include those which are related to the system being developed (i.e., the target system). We will discuss a model which shows representative development phases a system goes through. We will discuss the type of automated tools and techniques we believe are feasible, given the framework of ISDS/HOS.

5.1 Systems Preliminaries

System A (Figure 5.1.1) will be used to illustrate what we mean by a system. Let us consider System A as a function. When A is a function, we can consider the levels of A by decomposing it. In Figure 5.1.1, System A has been decomposed into two levels. Functions A_1 and A_2 are on the first lower level of A and functions B_1 and B_2 are on the second lower level of A.

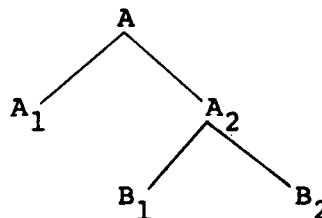


Figure 5.1.1: System A

The representative control map in the form of an invocation tree (Figure 5.1.1) shows the functions which will complete the specification for the execution requirements of A. The functions A_1 and A_2 are on the next most immediate lower level of A. When A is considered as data (i.e., a description of A) we refer to A as a layer instead of a function. When considered as data, the first immediate lower level of A (i.e., A_1 and A_2) is the data structure of A; similarly, B_1 and B_2 are the data structure of A_2 .

There are many trade-offs that must be considered in developing the layers of a system. They involve not only how many layers, but whether or not these layers are created statically or dynamically. We call those layers which are created statically (without execution of the target system) development layers. We call those layers that are created dynamically (during execution of the target system) execution layers.

Let us consider the problem of representing A in a form that is closer to a machine that will someday execute System A. In this case, we might need another outside system whose function is to convert A into a machine readable form. The name of this system is TRANSLATE (Figure 5.1.2). TRANSLATE is an example of a support system which may reside outside the realm of the target system.

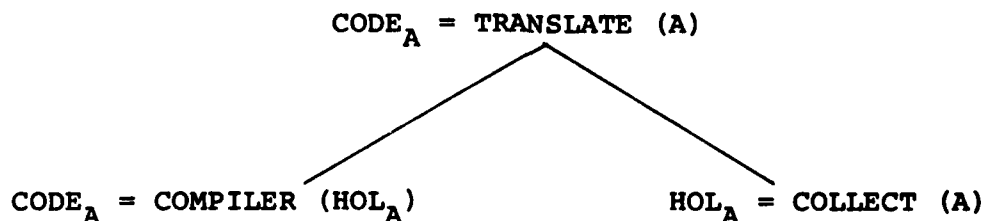


Figure 5.1.2: System TRANSLATE

Here, TRANSLATE chooses System A modules from a library in a higher order language form called HOL_A by means of function, COLLECT. TRANSLATE then produces machine code for the System A target

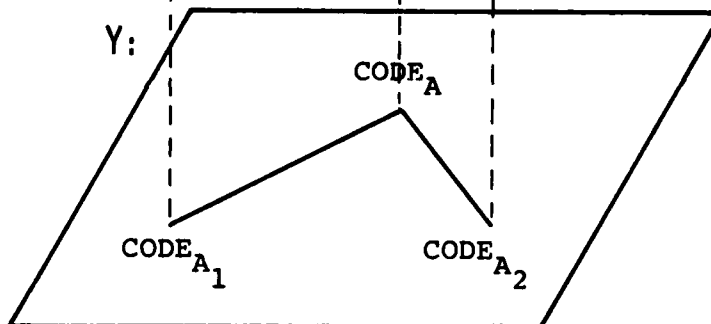
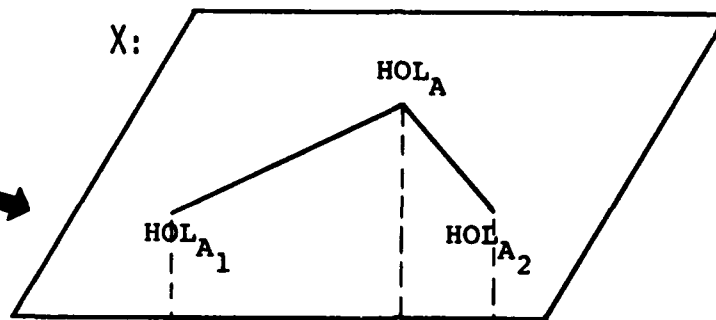
machine by means of function, COMPILER. This translation process is a static translation process. In this example, system HOL_A will never physically reside in the target machine when the target system is deployed; it is $CODE_A$ that will reside in the actual target machine. HOL_A will reside in the target system library within the host system library; COMPILER and COLLECT will reside in the host system library or in the host machine system itself.

A new layer, $CODE_A$, has been created by TRANSLATE. $CODE_A$ is a lower development layer of System A than HOL_A (Fig. 5.1.3). That is, $CODE_A$ is closer to executable form for the target machine than HOL_A .

In this example, we are treating System A as a target system and system TRANSLATE as a support system of System A. However, when TRANSLATE is the system being developed, it becomes the target system.

Some system support tools will reside in the target machine with the target system and dynamically produce "temporary" lower "development" layers of the target system as a result of a requirement from the target system itself. An example of such a function is an OS system or an interpreter which resides in the same machine with the target system application. These functions represent lower execution layers with respect to layer HOL_A . During execution, the request to execute the target system in a lower-layer state is relayed to the OS (Figure 5.1.4). The OS then creates the equivalent of a lower layer replacement (Figure 5.1.5) which temporarily resides in the computer until execution for that performance pass is completed. When this happens, the lower development layer disappears and the target system is once again resident in the target machine without its lower layer.

TARGET
SYSTEM
LIBRARY



TARGET
MACHINE

Y = COMPILER (X)

FIGURE 5.1.3: An example of a support system function COMPILER which creates a new layer for System A.

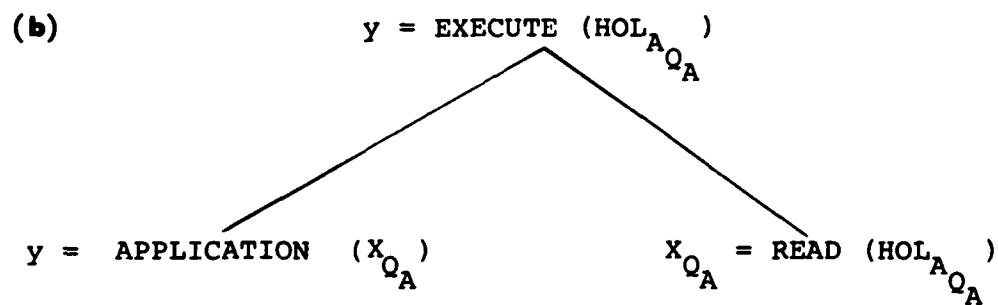
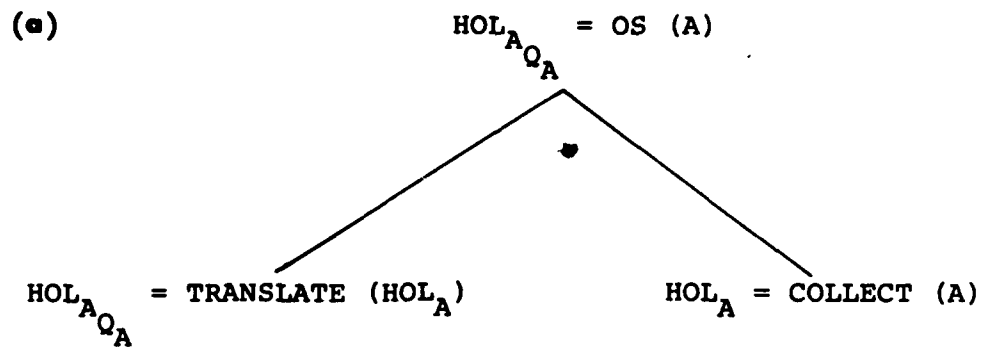


Figure 5.1.4: The OS system translates (a) and returns execution control back to System, A (b).

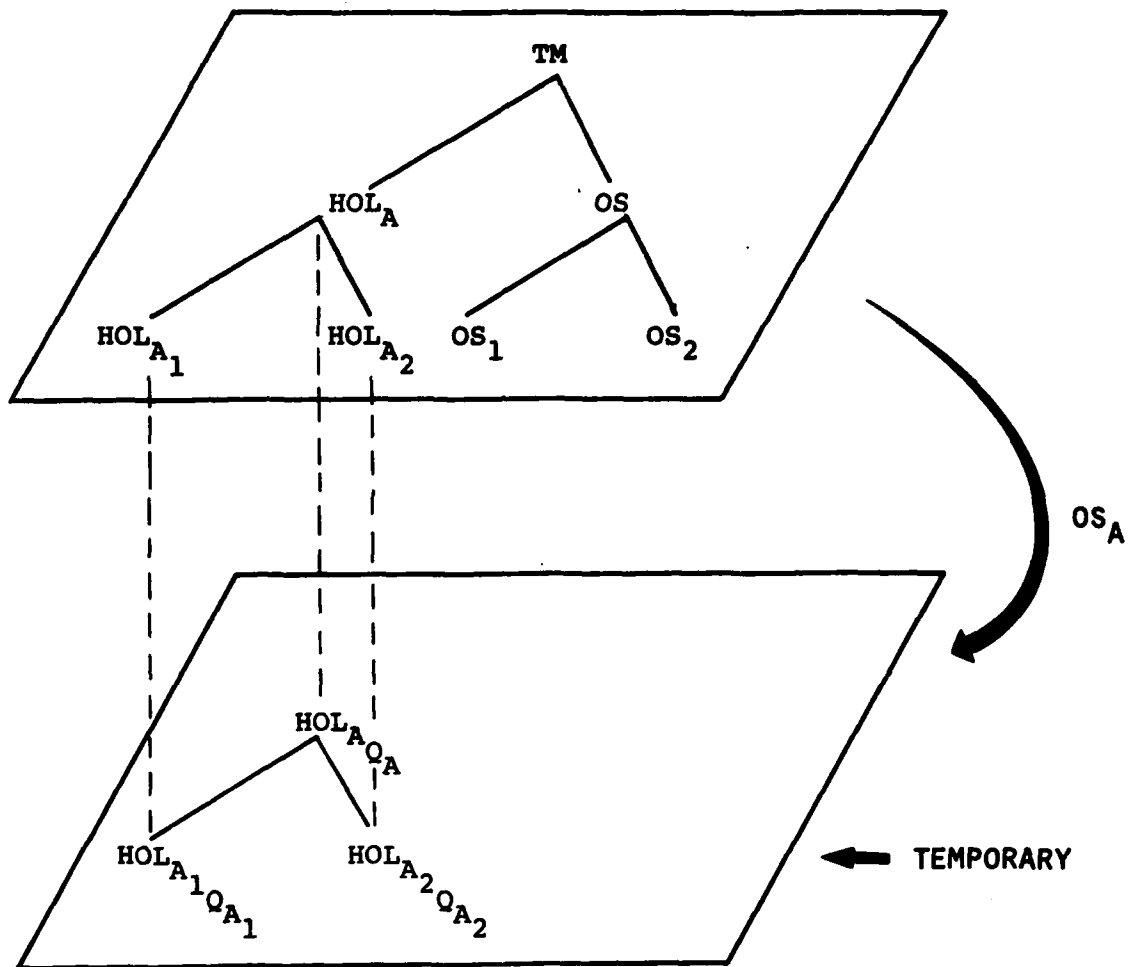


Figure 5.1.5: Dynamic Translation Process
(all in target-system machine)

It is important that a target system specification is consistent and complete. In order to define a system for completeness, it is helpful to know the requirements which include the characteristics of the various systems which influence the target system. Examples of these requirements are shown in Table 5.1.1. In order to define a system for consistency, it is helpful to know the many requirements to consider in defining the target system itself. Examples of these requirements are shown in Table 5.1.2.

TABLE 5.1.1

Requirements affecting the development of a target system

Customer	This includes the financial resources that are allocated and schedules that must be met for completing a system.
Mission	What is the job that is to be done and what is the environment within which the system is to be deployed? These requirements include nominal and off-nominal conditions.
User	What features are expected to be provided to the user of the system and how does the user interface with the system? (For example, does the customer expect faster turnaround time on an interactive system than the contractor expects?)
Tool	What tools are needed to develop the system throughout all phases of development? What tools are not needed, but would make the development of the system more cost effective? What are the requirements of these tools? What tools are already available and have been specified as off-the-shelf resources? How do these available tools affect the design of a system?
Methodology	How does the methodology affect the system and its development?
Development	What should the phases of development be? How many phases of development should there be? The more development layers there are in a system the more phases of development there might be to define and the more phases there are to define requirements for.
Host Facility	What are the requirements forced on the system by an existing host environment and its tools? What are the necessary requirements for the host facility to assist in the development of a system?
Completed Requirements	What modules are around that could be used to build a system? For example, if a subsystem is already completed, this might save development costs.

Support Systems	Each system that is developed has associated support systems. Examples of such systems might be an OS system to become part of the target machine, the environment system for a simulator, a data management system for keeping track of all the information about a system and its development; and a personnel system which manages a system. What are the requirements for those systems that are non-existent or those which require change? If these systems are not able to be changed, what requirements do they impose on the system being developed?
Design	What design requirements are enforced on a system design? For example, is it determined ahead of time that the system could be multi-processed, or parallel processed; that certain resources must be shared; or that error detection and recovery is required?
Standards	Certain standards force requirements on a system development. These include control structures and data types used to define the system; rules for decomposition; which tools to use for which process; approval forms for the request of a requirements change, anomaly reports; format for data-base descriptions of the content (type of information and level of detail) of requirements; development plans; official milestones; official meetings; official building process rules; approved hierarchy check points; numbered and titled memo series (for inside and outside of the project) for disseminating orders and information; reports; test plans.
Redundancy	These requirements might include the mean time between failure required of the target system, error detection and recovery, and back-up systems.
Testing	These include requirements for different levels and different types of official testing for a given module.
Statistical Gathering	These include the information that is gathered on all the development processes for a system. For example, information is gathered on anomalies, categories of anomalies, where anomalies are found, how anomalies are found, correction to anomalies, turn-around time for verification runs, CPU time, core size, costs, etc.
Overall	The ideal for all requirements is to be reliable, cost effective, and flexible both for development and during real-time.

TABLE 5.1.2

Requirements of the Target System

All systems have several categories of requirements as a stand-alone system. For example, consider System A (Section 5.1). Each development layer of System A inherit a set of requirements from the previous development layer.

Functional	In order to complete the specification of the execution of A, functions A_1 and A_2 are required. In order to complete the specification of the developmental layers of A, function TRANSLATE is required. In order to complete the specification of the execution layers of A, function OS is required.
Data	Certain data types and data structures are required to complete the specification of the execution of A. If, for example, A operates on a matrix, one data requirement for A is a matrix.
Performance	Performance requirements give limitations to input and output data. For example, the performance requirements of A might include a minimum and maximum range for the input values of x.
Documentation	Each layer of A has its own set of documentation requirements. The top development layer might have more comments for management than lower layers. A lower development layer contains more machine dependent information than a higher development layer. A layer described in AXES has AXES documentation requirements. A lower layer described in a HOL has its own documentation requirements. All layers of A should have a standard definition as to content of documentation. That is, some design documentation (like the name of a function) is a necessity for the system to run. Others may be necessary to convey useful information for testing purposes.
Resource Allocation	Each development layer of A is a resource allocation requirement for the next development layer of A (or on the next "machine"). The functions which translate one development layer of a system to another development layer (e.g., TRANSLATE in System A) provide resource allocation. Thus, a translation from the layer, A, in AXES to the layer A, in a HOL, could determine (from A in AXES) necessary and efficient resources.

5.2 ISDS/HOS Disciplines for Use in Developing a System

In a system which is solving the problem of developing another system (i.e., where DEVELOP is the function), we view requirements (R) and specifications (S) in the development system as input and output data respectively, as in (1):

$$S = \text{DEVELOP } (R) \quad (1)$$

In a development system, requirements are those items which are desired or needed; and specifications are the results which realize these requirements.

Every node of a system could be viewed as a development function. The subfunctions of a development function are phases. Thus, when a phase is viewed as a development function, its subfunctions are viewed as phases.

The function, DEVELOP, is decomposed in Figure 5.2.1. In decomposing a function, the designer makes use of the three ISDS/HOS primitive control structures discussed in section 4.0. In Figure 5.2.1 a phase of development is a subsystem of the DEVELOP system.

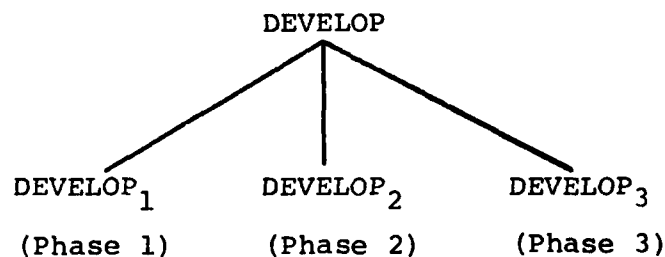


Figure 5.2.1

Each node in the development system is data used by the development discipline "machines" to realize the development function. These disciplines are design, implementation, verification, management, and documentation.

Later in this section, we will discuss, individually, the application of the design, implementation, verification, management and documentation disciplines.

In order to produce a specification, first a design process whereby one "conceives" and plans takes place. The implementation process realizes the plans which were conceived in the design process. A design process produces a set of requirements. The implementation process of that set of requirements produces a specification. That same implementation process is considered a design process with respect to the process that views that specification as a requirement.

The verification process verifies that the specifications fulfilled the requirements.

The management process directs the other discipline processes and ensures that all of these processes are carried out in a cost-effective manner and that the results of these processes are reliable. The manage function* either approves or disapproves its inputs which are outputs from one of the other disciplines.

* This function is provided for by an Assembly Control Supervisor (ACS). There is an ACS associated with each function in the development process, e.g. the ACS who corresponds to the manage discipline is a higher level ACS than the ACS for each discipline under his direction.

The documentation process records all of the outputs of the design, implementation, verification and management processes. If the process of design is manual, the documentation process is manual. If the process of implementation is manual, the documentation process is manual. If the process of verification is manual, the documentation process is manual; and, if the process of management is manual, the documentation process is manual.

In the case of a manual process, standard formats should be provided as to type and content needed to describe the outputs of that process. If, however, the design, implementation, verification or management process is automatic, the respective documentation for each process should only be produced automatically, since the output of each process is its documentation.

In Figure 5.2.2a, one step of the development process, where the input is requirements, R, and the output is specifications, S, as shown with their respective subsystems. For one step, the Design_Implementation disciplines, (Figure 5.2.2b), the Verification discipline, (Figure 5.2.2c), and the Management discipline (Figure 5.2.2d) are shown with their respective subsystems. In Figure 5.2.2, the documentation system (DOCUMENT) is included as subsystems of each discipline.

Note that verification discipline (Figure 5.2.2c) shows that verification is a reverse process to the design_implementation discipline, because in this process we are comparing the results to determine if the specifications meet the desired requirements instead of developing the specifications from the requirements. The design_implementation process not only produces preliminary specifications, but also produces test specifications for the verification process (Figure 5.2.2b). At the end of each process step, for all disciplines, documentation is produced to record the results.

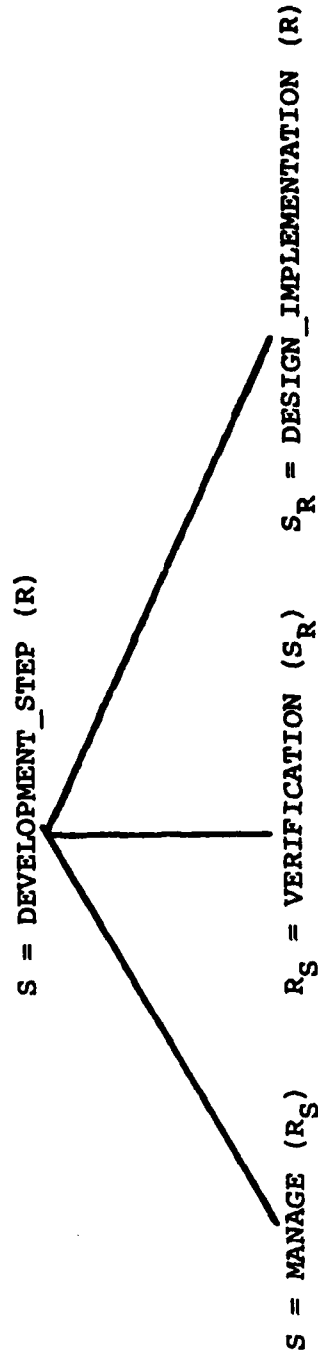


Figure 5.2.2a

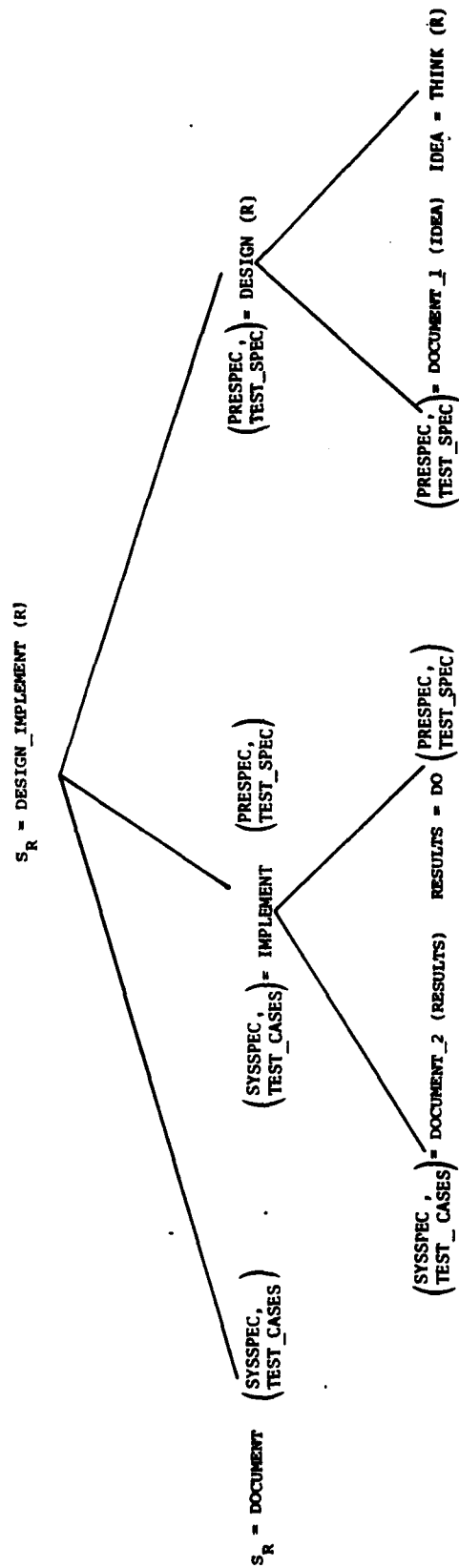


Figure 5.2.2b

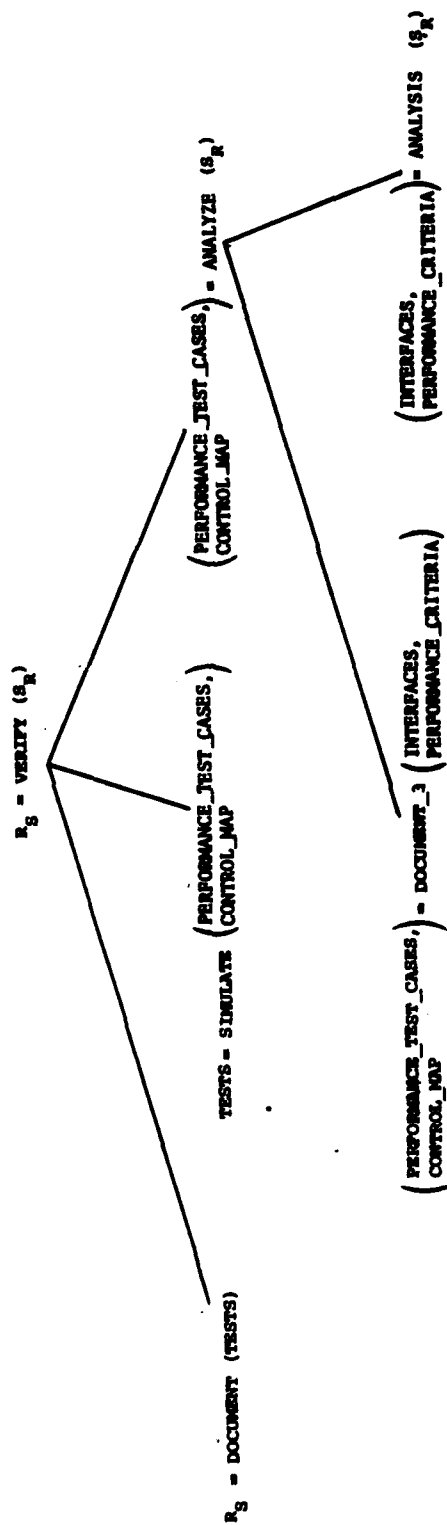


Figure 5.2.2a

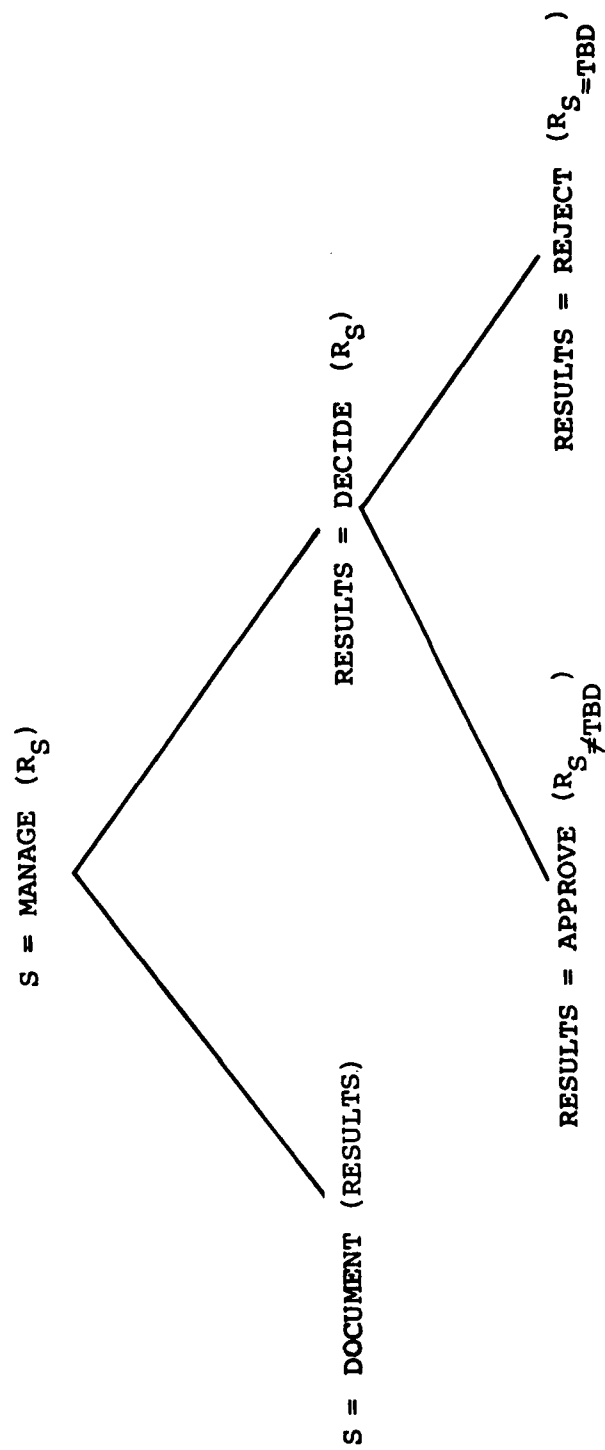


Figure 5.2.2d

To develop a given system, there are several successive and recursive steps of functions which provide a set of specifications from a set of requirements.

For example Figure 5.2.3*, one step of a typical management process is shown with an ISDS/HOS control map to illustrate the functions of a manager (which includes the design, implementation and verification disciplines). In this example, the manager determines if the requirements, R, are sufficient for his project. The documentation associated with each discipline in this step of development effects the output of each function in this example. In the manage process, the manager determines if all the proper forms have been filled out and if all the right testing procedures have taken place for a particular entry candidate. The manager checks the content of the forms and the requirements submitted. If the manager determines, along with experts in the areas relevant to this type of change, that the change is correct and its testing results are sufficient, the requirement or requirement change is approved by the signing of an official form. The new specification is then relayed by the manager as a new requirement for the next phase. If the requirements are not sufficient, he rejects them and starts a new step of the same development phase. This process continues until he is happy.

* In the examples that follow we refer informally to names of functions and their inputs and outputs. Thus, for example, when we refer more than once to DESIGN as a design process, it is not necessarily the identical function unless we specifically single out an equivalence.

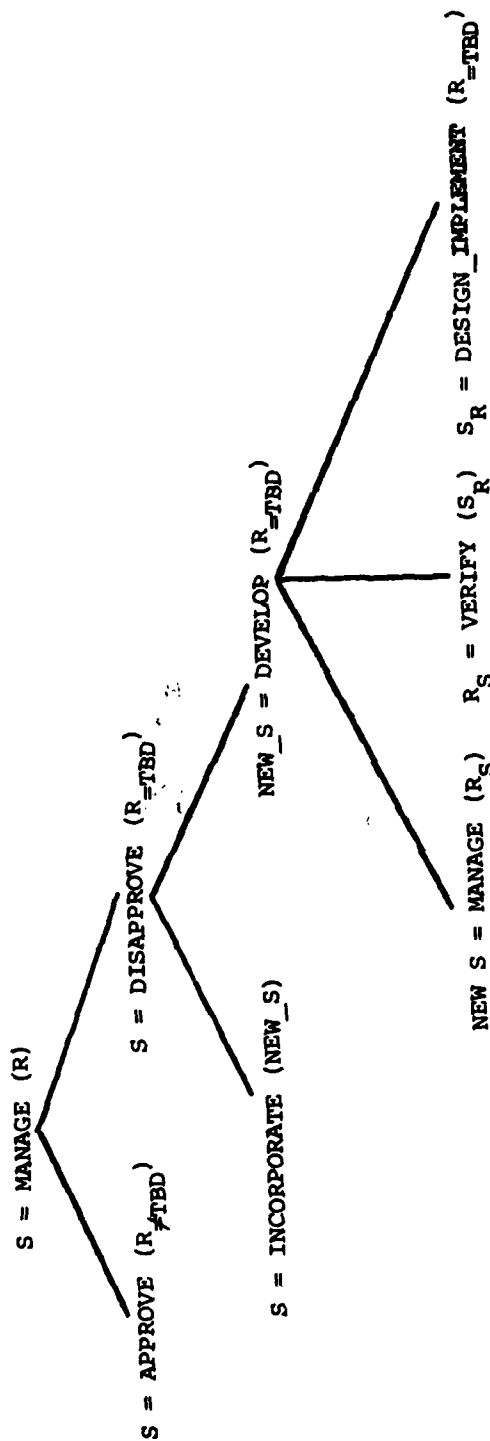


Figure 5.2.3: Example of management discipline between steps of one phase within an integrated system development process.

KEY	
TBD	- To Be Done
S	- Specifications
R	- Requirements
S _R	- Specification Prototype
R _S	- Requirements Prototype
NEW/R	- New Requirement
NEW/S	- New Specification

NOTE: R is requirements and S is specifications for a given phase.

For every development step of a given phase eq., (1) can be expanded to the following type of development function:

$$\overbrace{\text{Document(Results)}}^R = \overbrace{\text{MANAGE (Verify(Implement(Design(R))))}}^{\text{DEVELOP}} \quad (2)$$

All of the outputs of the MANAGE function and each nested function of MANAGE are the documentation of the development disciplines.

Figure 5.2.2 and Figure 5.2.3 are examples of system development disciplines which occur in any step of any phase of any system development. Thus, we have presented a template which illustrates the disciplines for application to the ISDS/HOS system development process.

In these examples, we have not yet singled out the particular tools that perform these functions. In some cases they are manual and in some cases they are automatic. The tools and their use is determined by availability or knowledge of availability at the start of a project development; the phase of development the project is in; and the decisions of project management. These template examples show only the type of functions that are needed to perform a development step and clarifies through standardization the patterns common to various concrete systems. In section 5.2, however, we do recommend tools for a system developed according to ISDS/HOS.

5.3 ISDS/HOS Development Phases for a System

Within ISDS/HOS there are only two development layers; i.e., the specification layer and the product which is the executable program. Because, however, not all the ISDS/HOS tools have been automated, other development layers would be required in the interim. We will discuss, here, a development process which includes transitional development layers that are developed manually. That is, we will assume that development layers go through translation processes which result in a description in a specification language, a resource allocation map, an HOL language and/or an assembly language and a target machine language. The target system OS could be either incorporated as part of the first execution layer of specification, a lower execution layer of specification or a lower development layer of the translation process (e.g., HOL layer). Thus, we will attempt to show alternative ways of developing a system with respect to its own OS.

In describing the development phases for a particular system we have chosen to relate ISDS/HOS to those phases which correspond to large DoD systems development (KOS75). For the purpose of flexibility (in case a project chooses a different breakdown of phase development) we will describe "templates" for various processes, tools or management techniques that could be used during each phase. In this way, substitutions can be made (e.g., if a project is too far along in development to replace or change already established methods; or if incremental introduction of a new methodology could be established where necessary for improvement of cost or reliability).

Our development model consists of four phases: (1) the Concept Formulation Phase (CF); (2) the Program Validation Phase (PV); (3) the Full-Scale Development Phase (FSD); and, (4) the Production and Deployment Phase (PD) (Figure 5.3.1).*

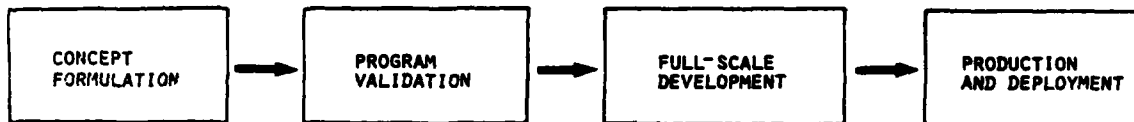


Figure 5.3.1: Four Major Phases of a System Development

Each phase receives requirements and produces specifications. These specifications are relayed to the next phase which in turn become the requirements to the new phase. In Figure 5.3.2, Phase_N receives R_N as requirements and produces the specifications, R_{N+1}. R_{N+1} is relayed to Phase_{N+1} as requirements.

$$\begin{array}{c}
 R_{N+1} = \text{PHASE}_N(R_N) \\
 \swarrow \quad \searrow \\
 R_{N+1} = \text{RELAY}(S_N) \quad S_N = \text{DEVELOP_STEP}(R_N)
 \end{array}$$

Figure 5.3.2: Delivery of Requirements for Next Phase

* The reader should be aware of the fact that these phases are sometimes associated with DoD funding divisions. This report is not intended to make such an association.

Throughout the development process of a system there are certain items which should always be made available to the various developers of the system. The same methodology and standards should be adhered to throughout. Common support systems, resources, tools, and modules should be made available if needed. In addition, certain new information is necessary to pass on from phase to phase. Sometimes quick turnaround information must be relayed as quickly as possible to relevant parties involved. Thus, the development management process should continuously facilitate all other development processes. In order not to overcomplicate the description of the individual target-system development process steps, we have chosen not to describe this process in all of the illustrated examples of the phases, but rather, we ask that the reader be cognizant of the fact that such a process is ongoing throughout a given system development. We will assume that all of the requirements passed on to a new phase from a previous phase include all of the items which later phases will need. For example, when a given phase turns over its specifications with all relevant information, libraries, etc. to the next phase as requirements, Phase_N deciphers what is needed to fulfill its own requirements for the target system development and maintains access to relevant information and resources (Figure 5.3.3).

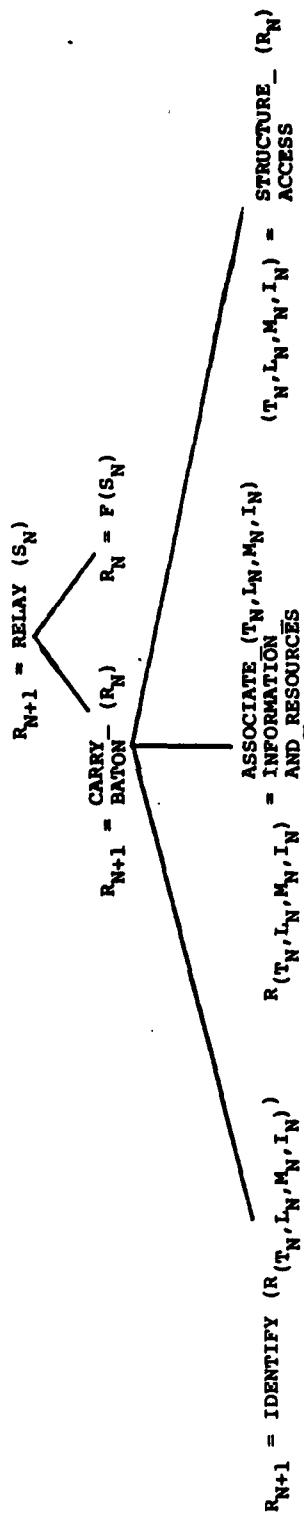


Figure 5.3.3: Example of System Management Techniques

KEY	
R	Requirements
M	Methodology, Standards
I	Information
L	Libraries
T	Target Requirements

Each phase turns over many requirements to the next phase. In addition there is always a new step of a phase which produces new sets of requirements for the next phase. Thus, it is possible for development activity to be doing on in all phases simultaneously. When partial specifications are completed by one phase, the next phase can begin to work on these partial specifications as partial requirements. Thus, one module could become frozen and delivered in the final phase before another one was ever formulated. Each requirement should be visualized as

$$R_{N+1} = \text{DEVELOPMENT}_N (R_N) \quad (3)$$

where eq. (3) can be replaced by

$$(r_1, r_2, r_3, \dots)_{N+1} = \text{DEVELOPMENT}_{(n_1, n_2, n_3, \dots)_N} (r_1, r_2, r_3, \dots)_N \quad (4)$$

A change in requirements to a system always results in new development steps (iterations). A change to one phase always forces an iteration of its phase and the phase immediately following it. Thus, if a change were made in the concept formulation phase, the CF phase would be directly affected (eq. (5)) and the program validation phase would be directly affected (eq. (6)). Likewise, the PV change would cause a change to the FSD phase (eq. (7)) which would cause a change to the PD phase (eq. (8)).

$$\text{CF}_{\text{STEP}} (\text{NEWR}) = \text{CHANGE} (R_1, \text{CF}(R_1)) \quad (5)$$

$$\text{PV}_{\text{STEP}} (\text{NEWR}) = \text{CHANGE} (\text{CF}(R_1), \text{PV}(R_2)) \quad (6)$$

$$\text{FSD}_{\text{STEP}} (\text{NEWR}) = \text{CHANGE} (\text{PV}(R_2), \text{FSD}(R_3)) \quad (7)$$

$$\text{PD}_{\text{STEP}} (\text{NEWR}) = \text{CHANGE} (\text{FSD}(R_3), \text{PD}(R_4)) \quad (8)$$

Many times a necessary change (due to an error or an adjustment needed to produce a specification) in requirements in one phase affects the requirements of an earlier phase. In this case the manager of each phase decides to change his own phase if it is determined that it does not affect the requirements of a previous phase; but he sends the change back to only the previous phase if the change falls in the category of a higher-level requirement. Therefore, it is possible that such a requirement change is sent back through all the preceding phases until it reaches the conceptual phase. In eqs. (9), (10), and (11) we show the possible requirements changes made to an earlier phase as a result of information found in a later phase.

$$CF_{STEP} (NEW R) = \begin{matrix} \text{ERROR} \\ \text{CHANGE} \end{matrix} (PV(R)) \quad (9)$$

$$PV_{STEP} (NEW R) = \begin{matrix} \text{ERROR} \\ \text{CHANGE} \end{matrix} (FSD(R)) \quad (10)$$

$$FSD_{STEP} (NEW R) = \begin{matrix} \text{ERROR} \\ \text{CHANGE} \end{matrix} (PD(R)) \quad (11)$$

Although only the previous phase receives official notice of a requirement error, the managers of each phase receive all error reports and notification of all requirements changes for each phase.

In Figure 5.3.4 and Figure 5.3.5 top level interfaces between development phases are shown with respect to the design, implementation and verification disciplines which take place in each phase.

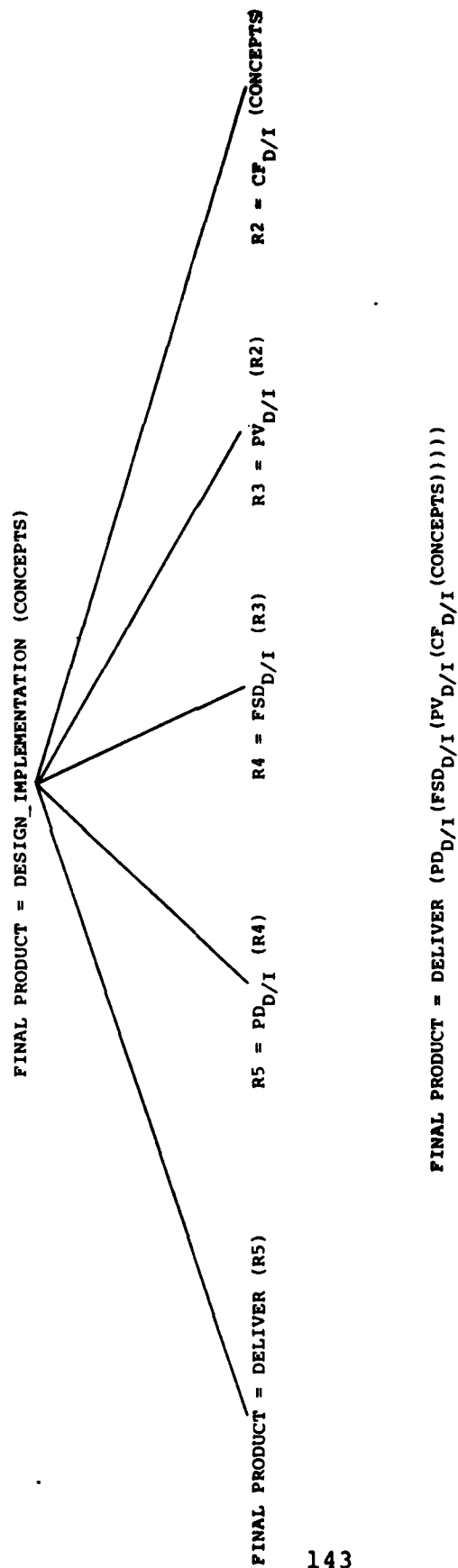
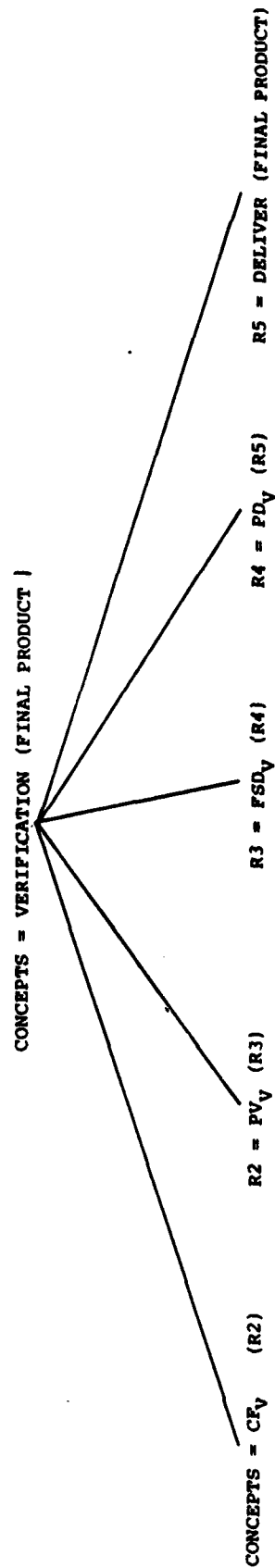


Figure 5.3.4: The Top-Level Design and Implementation Disciplines of an Integrated System Development Process



CONCEPTS = $CF_V(PV_V(FSD_V(PD_V(DELIVER(FINAL\ PRODUCT)))))$

Figure 5.3.5: The Top-Level Verification Disciplines of an Integrated System Development Process

5.3.1 Concept Formulation Phase

The first phase of our system development model is the concept formulation phase. In this phase, customer requests, mission requirements and top-level target system requirements are formulated. Requirements forced upon project management by support systems, developed target system modules or support tools which are determined by the customer as mandatory are formulated. In addition, support tools, support systems already developed, candidate target system library modules, or support tools which are determined by the customer and project manager to be available are considered as trade-offs and either incorporated or not incorporated into the formulations.

Once all of these requirements are well formulated, the designers use this information to design a preliminary control map for the target system.

The designer first jots down what he believes the functions to be in any order envisioned using whatever notation (English or otherwise) so desired. An attempt is then made to organize these functions hierarchically as a partial representative control map (an invocation map showing only function names may be sufficient at this time). After one or more preliminary design iterations, the partial map is complete enough to use as a guide for gathering more information. At this time the designer may have several questions he needs answered in order to make a complete control map. A complete control map will define and describe the functions and interfaces of the target system. The designer then writes down his questions for himself and/or others to perform interviews with the customer, various contractors and engineers involved in the project in order to determine more detailed contents of the requirements. A standard mechanism should be provided for recording the answers to these questions (that others or he, himself, have provided) in a data

base. This clerical process should be aided by standard forms or it could be automated.

After several iterations of data gathering and adding refinements to the control map, enough information is available for the target system designer to define additional standards. These standards can be formalized as specifications to define control structures or data types. These formal standards can be used to describe the formal specification of the system.

The formal specification of a system is described using the AXES specification language syntax and using the formal standards (AXES-defined data types and AXES-defined control structures).

The formal specification is automatically checked for interface correctness by a tool called the analyzer. Otherwise, a manual check of the specification is made for interface correctness. In addition, a manual check is made to see if the problem was defined as originally intended. The end result of the analysis could be either a small change or a restructuring of part of the system. This iterative process continues until the specification is a valid one in that it has complete and reliable interfaces. It is important to note here that it is possible for a specification to be a valid one and not do the job that the designer intended it to do.

The advantage of having the ISDS/HOS methodology to specify a valid specification is that the process of finding out why a specification does not produce the desired results is now limited to the question, "Did I specify the functions I really intended to specify?".

In Figure 5.3.1.1, an example of the concept formulation phase decomposition is shown. Table 5.3.1.1 shows the tools and techniques that correspond to the subsystems of the concept formulation system. Figure 5.3.1.2 is a more detailed decomposition of the concept formulation phase which demonstrates the iterative nature of the concept formulation process. Figure 5.3.1.3 demonstrates the interfaces within the SPECIFY system of the COMPLETE_SPECIFY system of Figure 5.3.1.2. Table 5.3.1.2 shows the tools and techniques that correspond to the subsystems of the SPECIFY system illustrated in Figure 5.3.1.3.

The output of the concept formulation phase is a set of specifications for the target system represented in AXES syntax, new standards defined with AXES, and a control map which shows graphically the decomposition of the functions of the system. In addition, the specifications for this phase include the original formulated requirements for the methodology and development processes as well as those requirements provided for support systems and support tools. All of these specifications are relayed to the next phase as requirements for the PV phase.

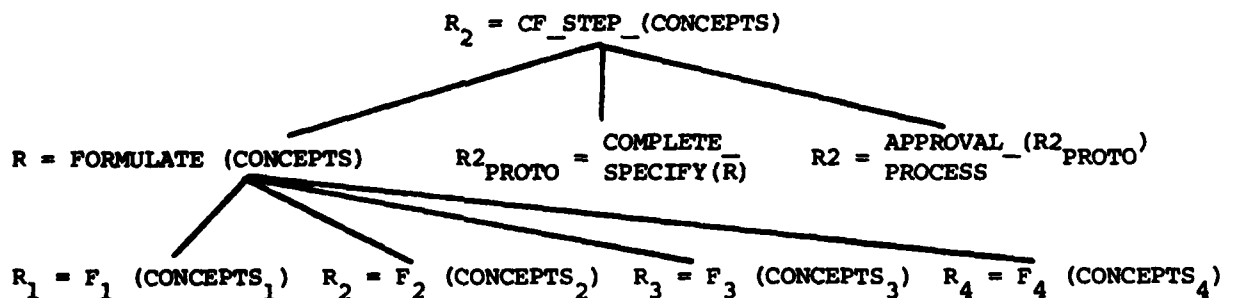
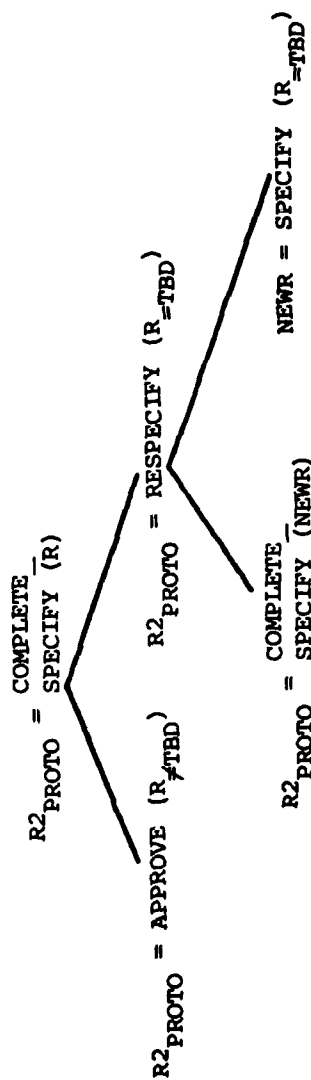


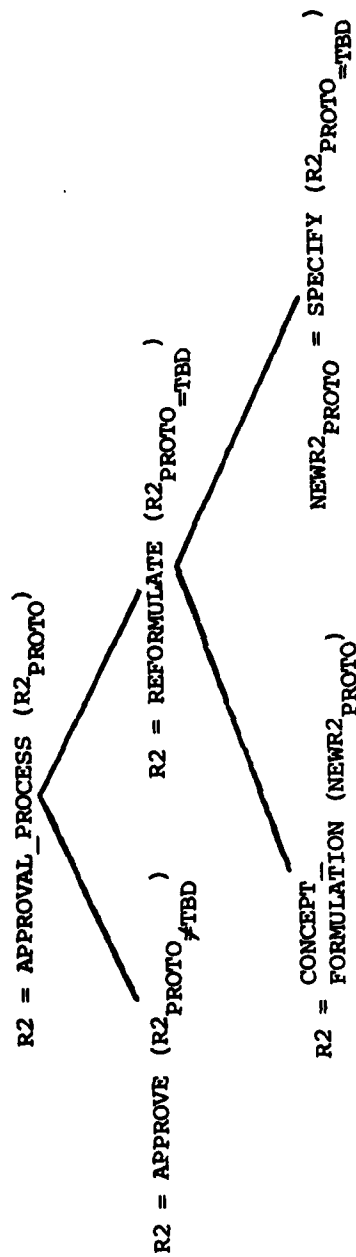
Figure 5.3.1.1: One Step of Concept Formulation Phase

TOOLS & TECHNIQUES	PROCESS	
ISDS/HOS Concepts	F ₁	Define standards for management structure, phases, building procedures and disciplines for system development.. Determine requirements for customer.
ISDS/HOS Concepts	F ₂	Determine <u>mandatory</u> available candidates for support tools, target module, and support systems available now for use in this and later phases.
ISDS/HOS Concepts	F ₃	Determine mission requirements for conceptual inputs. Specify standards for describing target system functions. Specify system functions and determine those which are already defined.
Manual and Data Management System	F ₄	Determine preliminary resource allocation needs from system resources that are <u>mandatory</u> and/or available for later allocation (e.g., sensors, host and target computers, resident software).
ISDS/HOS Management Concepts	APPROVAL PROCESS	The CF manager approves CF specifications as a deliverable for CF process.
(See SPECIFY Figure 5.3.1.3)	SPECIFY	The formulation of the target system requirements.

Table 5.3.1.1: Concept Formulation Phase Tools and Techniques



(a) More Than One Step of the Specify Process Within the Concept Formulation Phase



(b) More Than One Step of the Complete Concept Formulation Phase.

Figure 5.3.1.2: Examples of Iterative Steps Within the Concept Formulation Phase

TOOLS & TECHNIQUES	PROCESS
ISDS/HOS Decomposition Concepts	JOT - jot down notes about functions and function names that are in target system, organize and reorganize the notes until they are existent in a hierarchical form to work with (i.e., attempt to formulate a representative control map).
ISDS/HOS Decomposition Concepts;	PLAN_1 - complete as much as possible a commented control map with questions which reflect access rights, ordering, invocation specification properties of target system functions.
ISDS/HOS Decomposition Concepts; Data Management System; Standard Forms recorded manually or automatically incorporated with problem statements format into data base; Text Editor.	INTERVIEW - use the control map, comments and questions to interview system, support system engineers and customer to fill in the missing parts of the control map. Enter your own original information and information acquired from questions on standard forms into the requirements data base.
AXES Abstract Control Structures, Abstract Data Types, Data Management System Text Editor	PRODUCE_SYSTEM - (1) define and design new abstract control structures or data types or standard methods of expressing requirements in the specification using AXES created statements*; (2) transcribe data base information about control map to AXES statements; (3) write a complete narrative for information purposes; and (4) initiate top-level test specifications for program validation performance constraint testing.
Analyzer (manual or with analyzer) Text Editor Formatter	TRANSLATE_SIM - analyze AXES statements, standards for interface consistency (proper decomposition) format AXES listing, standard listing and English narrative listing. A control map is produced which notates and describes interface errors if not correct or which shows functions and interfaces if correct. Manual check to see if problem is defined as originally intended.
Manual, Data Management, Approval Forms, Sign-Off Procedures, Text Editor	APPROVE - go through ACS approval channel for acceptance for a frozen specification module.

* Some preliminary standards are already chosen.

Table 5.3.1.2: Concept Formulation Phase Specification Process Tools and Techniques

5.3.2 Program Validation Phase

The Program Validation (PV) phase concentrates on the detailed analysis of the functions defined in the Conceptual Formulation phase. At this time system performance constraints are considered in the design process of completing a more detailed specification. Timing and accuracy analysis is performed. Restrictions imposed by known environments of the target machine are studied. Trade-offs for reliability are performed. Included are considerations of fault tolerance, error detection and recovery, customer needs and security requirements. These studies are performed either manually or automatically by simulation performance testing. The design discipline in this phase is very similar to the design discipline in the CF phase. That is, after several iterations a revised control map is formulated along with a revised data base reflecting all the changes and additions made to the target system.

Throughout this process, the simulator requirements may also need to be updated to reflect the changes to the target system requirements. After the revised control map has been verified, the top layer resource allocations that have not been associated with target subsystems are determined, i.e., determinations are made as to which functions will be executed in a hardware, software or firmware "machine".

If prior resource allocations have not been determined, the complete control map now exists in the form of one development layer described and verified in AXES statements. Without the resource allocation (Figure 5.3.2.1, System AXES_s illustrated with only its function names) there is flexibility with which to build the development layers, since this same system could now be used with various alternative resource allocations.

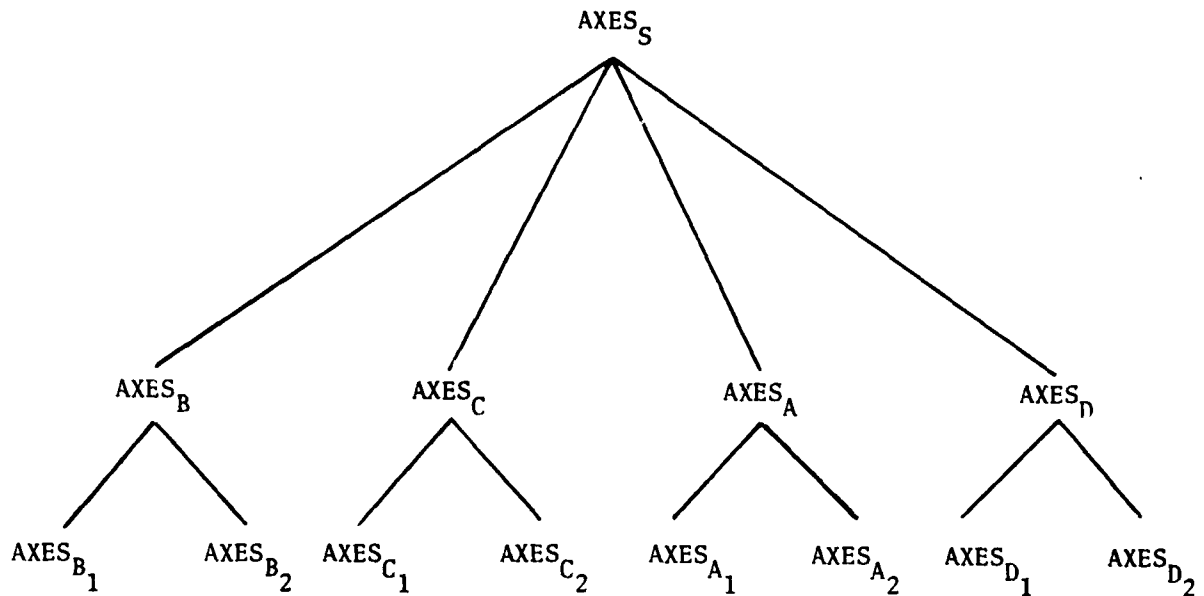


Figure 5.3.2.1: Target System $AXES_S$

Top level resource allocations are manually assigned to $AXES_S$. This process is a similar process to choosing top level managers or top level functions of a system. System $AXES_S^{HARDWARE/SOFTWARE_S}$ is an example of a system where specifications include the top layer resource allocation decisions (Figure 5.3.2.2). Here, the target system $AXES_S$ has been assigned to reside in a computer-based system. $AXES_C$ has been assigned to reside in a sensor system. $AXES_A$ has been assigned to reside in a target computer, etc. Later, $AXES_{A_1}$ might be assigned (manually or automatically) to software of the computer system; $AXES_{A_2}$ might be assigned to firmware of the computer system.

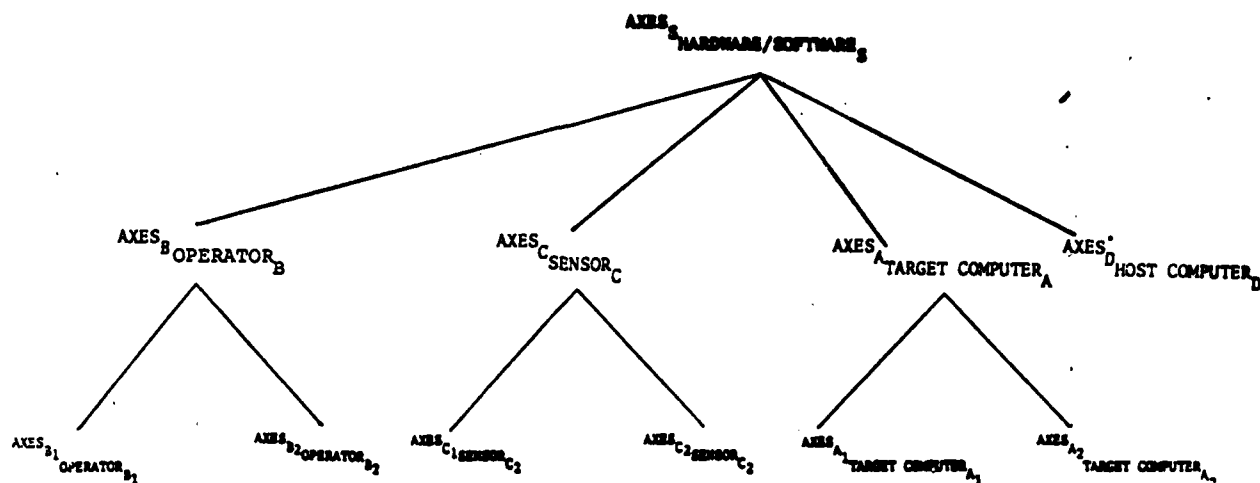


Figure 5.3.2.2: An Example of Top Layer Resource Allocation for Target System AXES_S

Once the top layer resource allocation choices have been made, support systems to the target system and support tools for the various subsystems can be determined. Some of these support systems and support tools may be selected from an existing library of support systems and support tools. Thus, an existing OS system might be chosen as a support system which will reside with the target system in the target machine when it is deployed. An existing HOL or existing compiler might be chosen as off-the-shelf tools which will not reside in the target machine when it is deployed.

Once the tools have been selected, those that do not exist or only partially exist should have the development of their own systems completed. The priorities and milestones for these systems development should, of course, take into consideration the milestones of the development of the target system. Each of the support system developments should proceed with a development process similar to the one used for the target system described here, since these support systems are target systems with respect

to their own development processes. The processes that can take place within the Program Validation Phase are illustrated in Figure 5.3.2.3. Table 5.3.2.1 is included as a description of the functions of Figure 5.3.2.3.

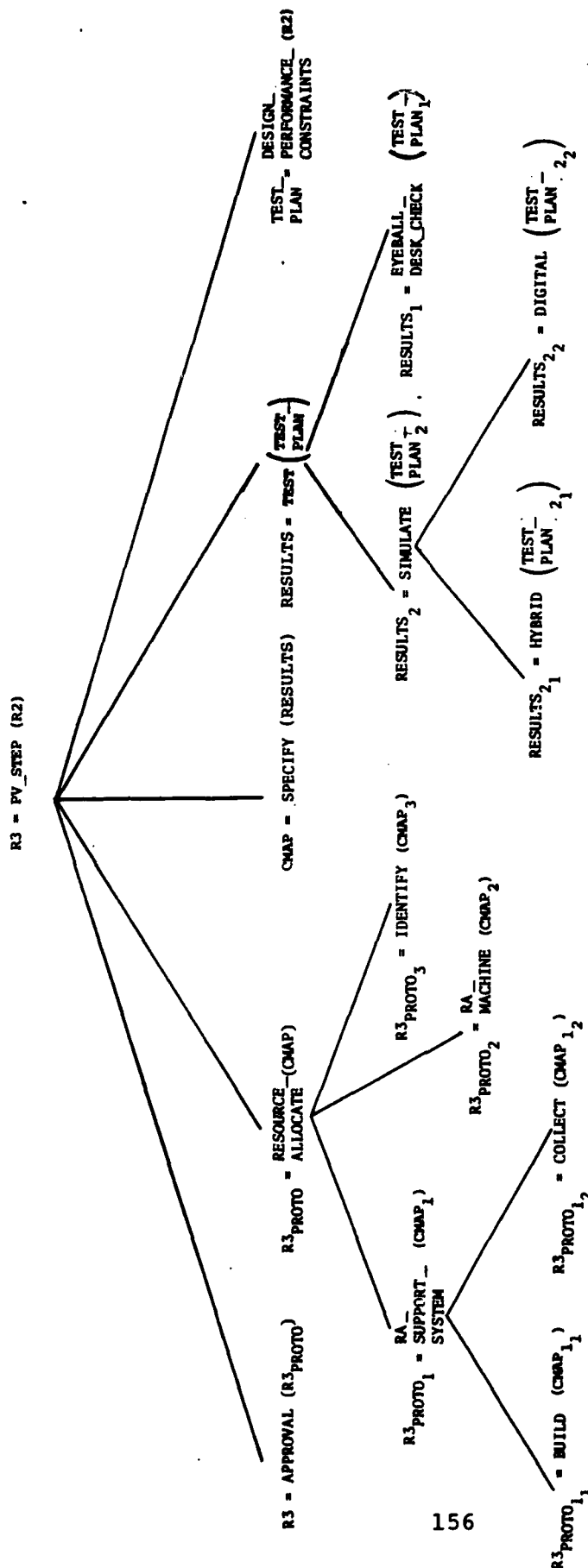


Figure 5.3.2.3: One Step of the Program Validation Phase

KEY	
PV	- Program Validation Phase
C2	- Requirements for Phase 2
C3	- Requirements for Phase 3
RA	- Resource Allocation
CMAP	- Control Map
PROTO	- Prototype

TOOLS & TECHNIQUES	PROCESSES
Manual, Data Management System	DESIGN PERFORMANCE CONSTRAINTS - perform analysis of timing, accuracy, environment restrictions, error handling, fault tolerance, security and customer needs. Design test plan for trade-off studies and analysis
Manual	EYEBALL DESK CHECK - static and manual analysis and verification.
Digital simulator, Hybrid simulator, Text Editor, Performance Monitor	SIMULATE - dynamic analysis and verification
AXES, Analyzer, Data Management System, Narrative Updater	SPECIFY - redesign control map incorporating performance constraints narrative, etc. this is a new step of the CF phase, function SPECIFY (Figure 5.3.1.3).
Manual or Data Management System Collector	IDENTIFY - select control map information needed to describe target system.
Manual, Data Management	RA MACHINE - perform top layer resource allocation.
Manual, Collector	COLLECT - collect available support systems needed to develop target system.
ISDS/HOS Concepts	BUILD - begin building support systems needed to build target systems.
Data Management, Approval Forms, Sign-off Procedures, Text Editor	APPROVAL - ACS supervisor approves PV Phase requirements.

Table 5.3.2.1: Program Validation Phase
Tools and Techniques

5.3.3 Full Scale Development Phase

The purpose of the Full Scale Development (FSD) phase is to translate all of the requirements for the target system into a form which is able to be interpreted for execution on a target machine. Ideally, these requirements exist in a control map form, are described in AXES, have been analyzed by the analyzer for correctness of interfaces; and have been analyzed by a means such as simulation to verify that certain performance criteria have been incorporated into the system definition.

The development process of the FSD phase is shown in Figure 5.3.3.1. During the FSD phase, iterative steps take place if there is a change to be made in the requirements. If the change only affects the FSD itself (i.e., an error was made in the FSD phase), the ACS of the FSD phase decides to fix his own intermediate requirements (CHANGE_THIS_PHASE_ONLY). If the change is necessary due to a problem resulting from a previous phase, the ACS of the FSD phase officially notifies the ACS of the PV phase (PROGRAM_VALIDATION).

A step of the FSD phase is shown in Figure 5.3.3.2a. This step includes two major resource allocation efforts, RESOURCE_ALLOCATE and RAT_I. Table 5.3.3.1 is included to describe the functions of Figure 5.3.3.2. RESOURCE_ALLOCATE illustrates the process of selecting target "machines". (If the target machine is not selected, an iterative resource allocation process takes place.) Several trade-off studies for timing and memory considerations are made. A candidate target machine is then selected and the process of optimally allocating the system to that machine takes place. The function, RAT_I, provides resource allocation within selected machines. After the target machine

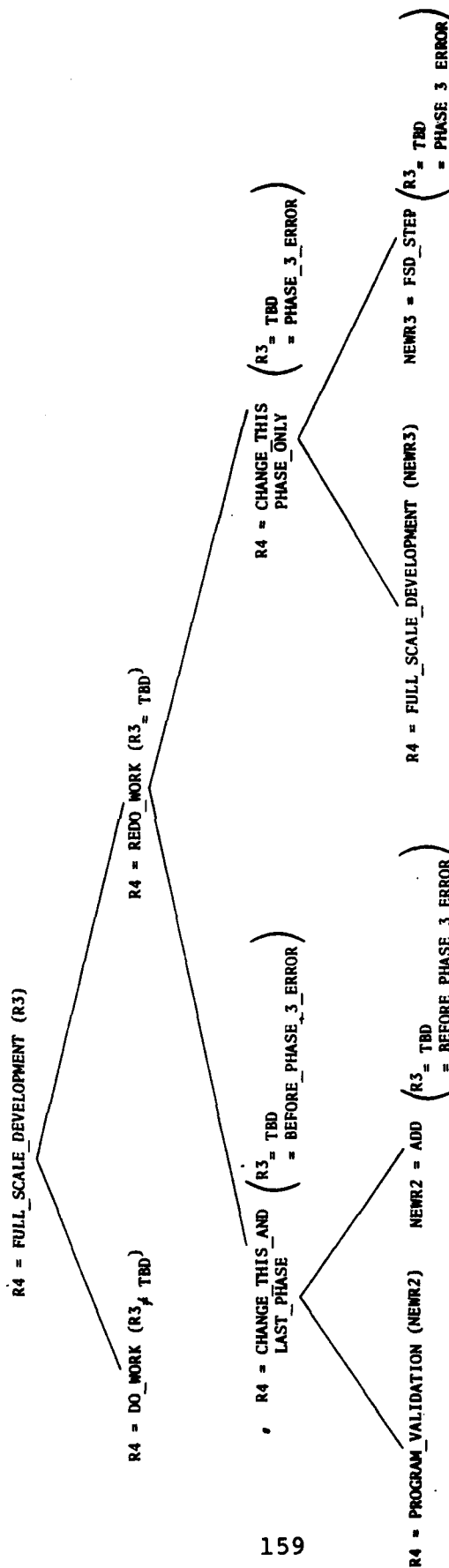


Figure 5.3.3.1: Potential Iterative Steps Within the FSD Phase

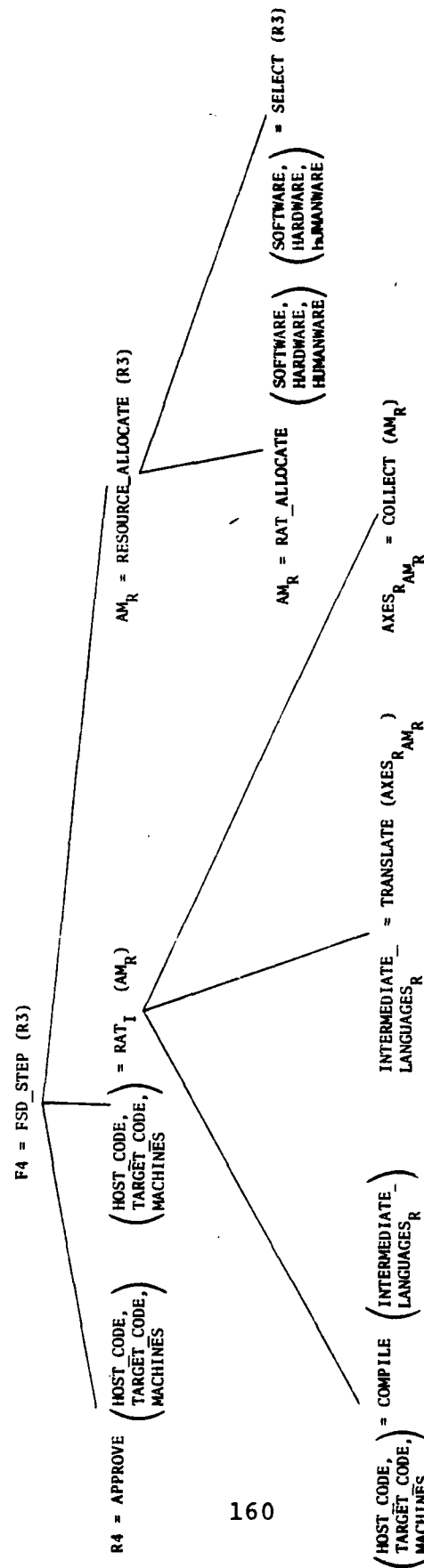


Figure 5.3.3.2: An Example of One Step of the Full Scale Development Phase (For the Incremental Model)

TOOLS & TECHNIQUES	PROCESSES
manual, simulator, structuring executive	SELECT - Select those requirements which go in software, hardware and humanware machines. These requirements all exist in AXES form. The OS modules for the target machines are included in these requirements.
RAT, manual analysis (if RAT is manual) simulator	RAT_ALLOCATE - perform a resource allocation process which is independent of the target machine that the target system will execute on. Verify these results:
COLLECTOR	COLLECT - Collect the requirements in architectural form.
manual writer or automatic translator (either a post-processor for an analyzer or pre-processor for an existing HOL), automatic structured design diagrammer.	TRANSLATE - Translate the requirements to an intermediate form(s) such as an HOL, assembly language, macros, etc. produce an execution map (structured diagram).
COMPILER, ASSEMBLER, simulator, manual analysis if translate process is manual	COMPILE - Compile intermediate language to code for target machine, host machine, verify the results.

Table 5.3.3.1: Tools and Techniques Applied
Within One Step of the FSD Phase

is selected, RAT_I is concerned with optimally allocating the target system to fit the resources of that machine.

In Figure 5.3.3.2, RAT_I is decomposed for the incremental ISDS/HOS development model whereby resource allocation is not completely automated.

The system and its subsystems are then translated into a form which is ready to be executed on the target machine. Until the target machine is built, the target machine can be simulated in a host computer with the simulated environment of the target system. In this way the target system can be simulated in order to verify that performance requirements are met by the system.

We envision for ISDS/HOS that the FSD phase will become more straightforward than it is today; since it should be possible to automatically translate a functional requirement (described in AXES and analyzed by the analyzer) direct to its target machine coded form. This translation process would be performed by the RAT. (Some of the resource allocations are made statically and some are saved for dynamic allocation). We would perform the major steps as indicated in Figure 5.3.3.3 for each target system.

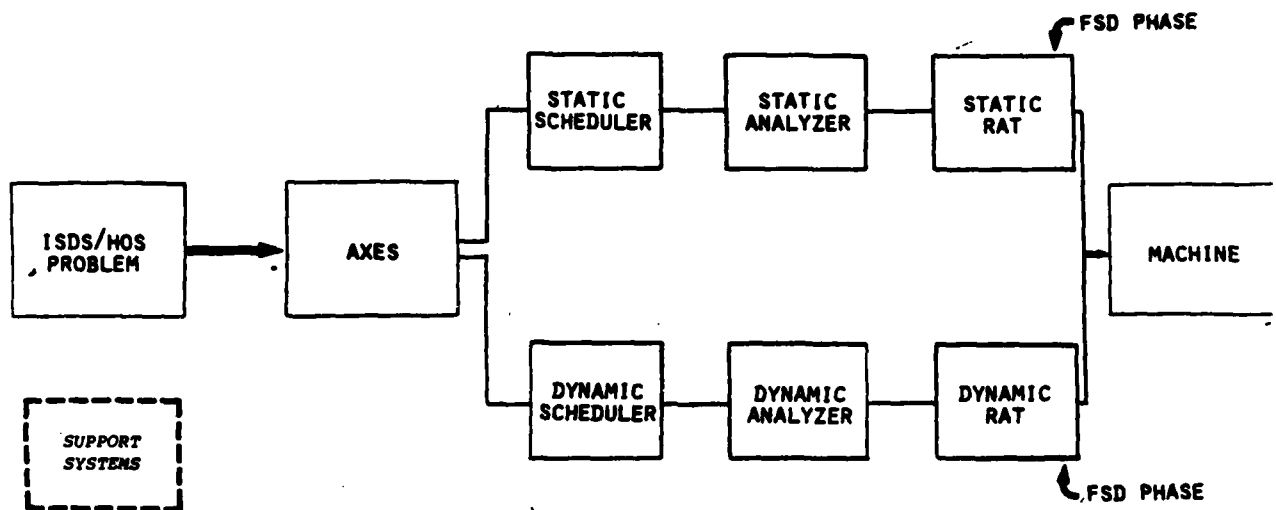


Figure 5.3.3.3: Major Translation Steps of an Integrated System Development Process

Until the tools which will automate the FSD phase are available, we must consider alternatives for the FSD phase. There are many ways in which a system in AXES could be translated to a target machine. The various translation processes and alternate combinations of these processes are sometimes performed manually and sometimes performed automatically. Some of these translation processes are static.

- Assembler - translates assembly code (and sometimes interpreter code) to machine code.
- Analyzer - translates from a language form to a control map form.
- Writer - translates from one language to another language (e.g., convert from AXES statements to HOL statements).
- Compiler - translate from a language to a lower level code that is closer to the "machine layer".

- Collector - collect from a library several subsystems to form a system.
- RAT - translate unlimited resource allocations to limited resources for target machine.

Some of these translation processes are dynamic:

- Interpreter - interprets in real time higher layer code and temporarily creates lower layer code for a given execution pass of each higher level code statement.
- Dynamic Analyzer - translates a target system in real time to a temporary control map.
- Dynamic RAT - allocates resources to a process in real time.
- Dynamic Scheduler - interprets in real time requests of the target system to be executed and temporarily creates a lower layer for the target system until a given process is executed.
- Dynamic Collector - collects and gives priorities to subsystems in real time.

Various combinations of static and dynamic translation processes can be collected for a particular target system development. Suppose we deploy a target system, A, with a resident OS. Both system A and system OS requirements have been received by the FSD phase in the form of AXES statements, R. The ACS of the FSD phase decided that the requirements for both system A and system OS should be translated to an HOL for implementation. In this case the translation process to the target machine might proceed as follows: (see Figure 5.3.3.4).

$$(\text{CODE}_{\text{OS}}, \text{CODE}_A) = \text{COMPILER} (\text{WRITER}(\text{COLLECT}(\overline{A_R, \text{OS}_R}))) \quad (12)$$

Here, the translation process which occurs in the target machine would be represented by

$$Y = \text{OS}(A(x)) \quad (13)$$

Other alternatives for the set of translation processes for the FSD phase of System A are:

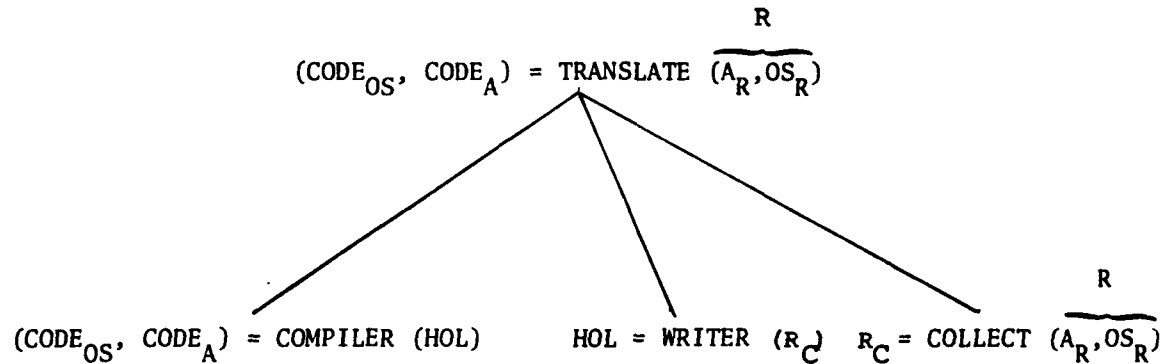


Figure 5.3.3.4: Translation Process for System A

$$\text{CODE} = \text{ASSEMBLER } (\text{WRITER } (\text{COLLECT } (\text{A}_{\text{R}}, \text{OS}_{\text{R}}, \text{INTERPRETER}_{\text{R}}))) \quad (14)$$

where, in the machine $y = \text{OS } (\text{INTERPRETER}(\text{A}(x)))$;

$$\text{CODE} = \text{ASSEMBLER } (\text{WRITER } (\text{COLLECT } (\text{R}))) \quad (15)$$

where, in the machine, $y = \text{A}(x)$;

$$\text{CODE} = \text{COMPILER } (\text{WRITER } (\text{RAT } (\text{COLLECT } (\text{A}_{\text{R}}))) \quad (16)$$

where, in the machine, $y = \text{A}(x)$.

When we have the ISDS/HOS tools automated, the recommended translation method for the FSD phase would be to proceed as follows:

$$\text{CODE}_A = \text{RAT} \left(A, \overbrace{\text{OS, MOS}}^R \right) \quad (17)$$

where, in the machine,

$$y = \text{MACHINE} (\text{MOS}(\text{OS}(A(x)))) \quad (18)$$

is the functional specification for executing System A where OS (represents the machine independent operating system functions), MOS (represents the machine dependent operating system functions) and machine are execution layers of CODE_A ; MOS is an execution layer of OS; etc.

The recommended development and execution layers of a system are illustrated in Figure 5.3.3.5. The function described in Figure 5.3.3.5 is

$$\text{TM} = \text{F}_{\text{RAT}} (\text{L}) \quad (19)$$

where

$$\begin{aligned} \text{CODE}_{\text{MACHINE}} (\text{MOS}(\text{OS}(A(x)))) = \\ \text{F}_{\text{RAT}} (\text{AXES}_{\text{MACHINE}} (\text{MOS}(\text{OS}(A(x))))) \end{aligned} \quad (20)$$

F_{RAT} is the functional specification for translating target machine systems to an executable form; e.g., CODE_A is a development layer of AXES_A ; CODE_{OS} is a development layer of AXES_{OS} , etc.

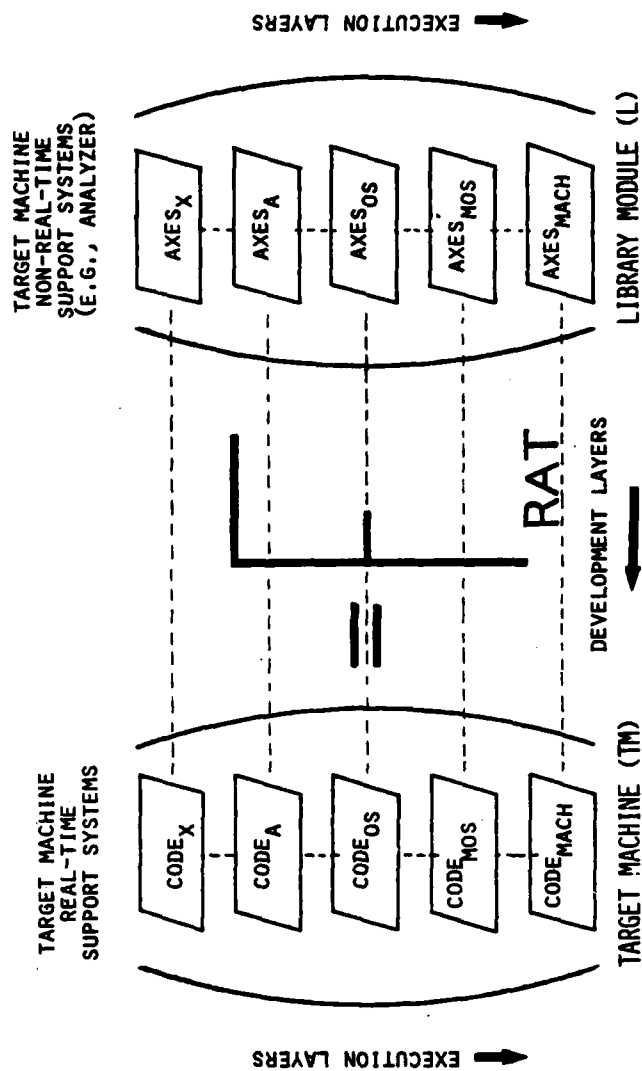


FIGURE 5.3.3.5: DEVELOPMENT AND EXECUTION LAYERS OF SYSTEM A

With this recommended method, the requirements (stated in AXES) for the target system, the independent of the machine OS support system, and the dependent on the machine OS support system can all be translated directly to the target machine code. Changes to each of these systems can be made without affecting each other. The target system can be transferred to another machine system, the independent of the machine OS support system can be transferred to another machine system; and the dependent on the machine OS can be changed separately to reflect changing machine requirements.

5.3.4 Production and Deployment Phase

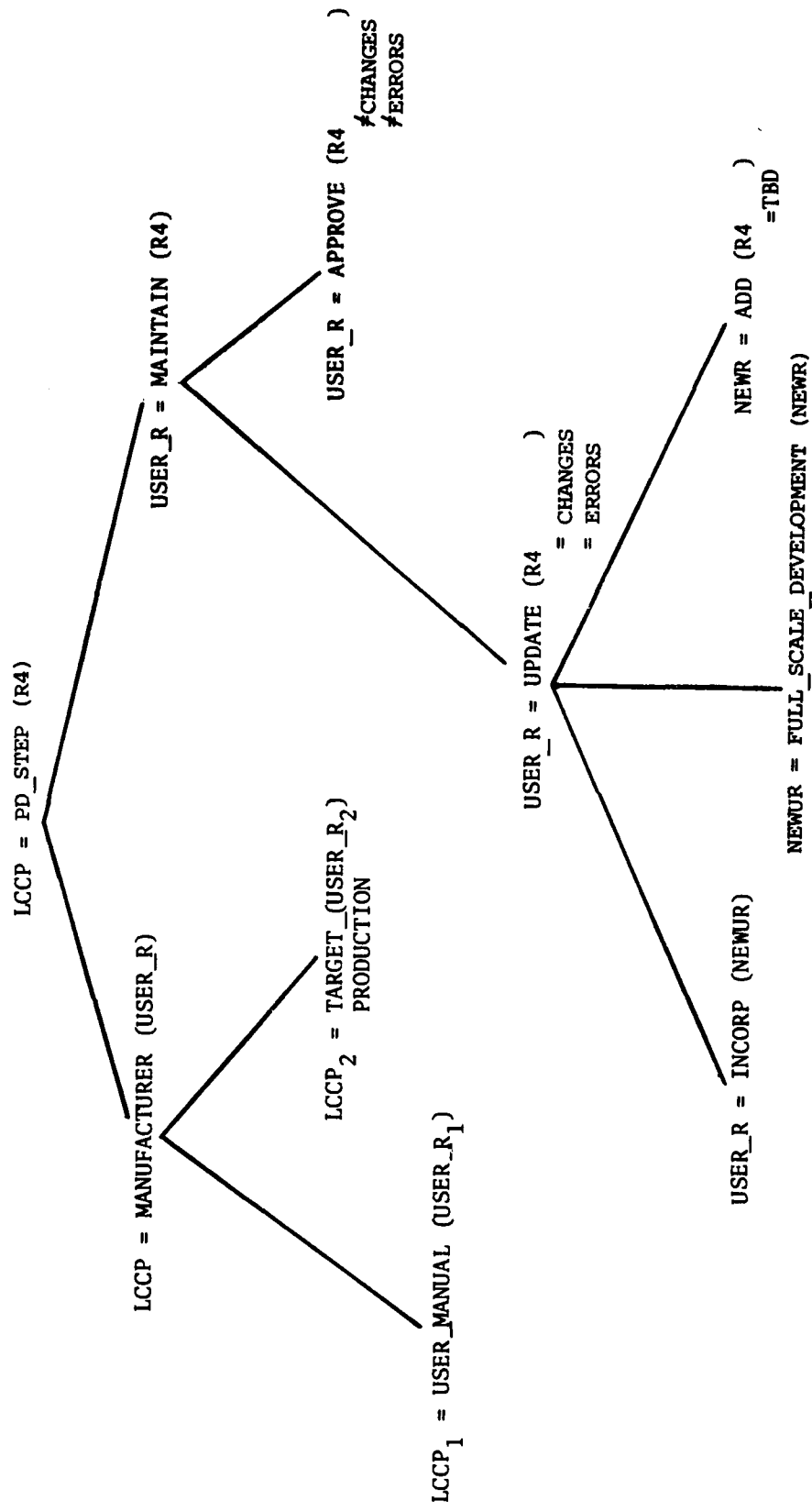
During this phase, efforts are concentrated on preparing manuals and instructions for the use of the target system that has been developed. User requirements were initially stated before the CF phase and formulated in the CF phase. The user manual should reflect these functional requirements but it should also contain additional information which describes the developed and working system.

The target system is now operational. Often, many changes are requested as a result of operating the target system in its real environment. These changes are due to improvements desired by the user or errors found in the use of the system. All errors should be treated as if they were a change to the FSD phase requirements and are reported to the ACS of the FSD phase. All new user requests are treated as new requirements and are reported to the ACS of the CF phase, the ACS of the PV phase and the ACS of the FSD phase. All three managers must **officially** approve the introduction of a change.

Once all the changes have been incorporated, the manufacturing of the target system is completed and the target system is deployed into the field as a delivered product. Figure 5.3.4.1 shows an example of processes that might take place in the Production and Deployment phase.

5.4 Tools Used During the Phases of System Development

Table 5.4.1 summarizes all the tools used during all phases of system development. These tools will be further described in Chapter 6.



KEY	
LCCP	- Life Cycle Complete Product
PD	- Production and Deployment

Figure 5.3.4.1: An Example of Processes of the Production and Deployment Phase

TOOLS	CONCEPT FORMULATION	PROGRAM VALIDATION	FULL-SCALE DEVELOPMENT	PRODUCTION AND DEPLOYMENT
<u>COMPONENT TOOLS</u>				
AXES	*	*	*	*
ANALYZER	*	*	*	*
STRUCTURING EXECUTIVE			*	*
STATIC RESOURCE ALLOCA- TION TOOL			*	*
<u>SUPPORT TOOLS</u>				
DATA-BASE STRUCTURE	*	*	*	*
RESOURCE MONITORING	*	*	*	*
INTER-REVISION UPDATER	*	*	*	*
COLLECTOR	*	*	*	*
TEXT EDITOR	*	*	*	*
TEXT FORMATTER	*	*	*	*
SIMULATOR		*	*	*
EMULATOR		*	*	*
PERFORMANCE MONITOR		*	*	*
<u>INCREMENTAL TOOLS</u>				
ASSEMBLY LANGUAGE			*	*
MACRO-PROCESSOR/ASSEMBLER			*	*
HIGHER ORDER LANGUAGE			*	*
COMPILERS			*	*
STRUCTURED DESIGN DIAGRAMMER			*	*
INTERACTIVE DEBUGGER			*	*
INTERPRETER			*	*

Table 5.4.1: Tools Used During the Phases of a
System Development

5.5 System Building Process

The ISDS/HOS building process is an orderly technique for "freezing" system modules. Table 5.5.1 demonstrates a system building matrix used by management personnel to track the development of a system. Each subsystem can be tracked by the elements of the system building matrix. In Table 5.5.1 each building layer of system functions are tracked separately as an integrated system building layer. With respect to a given system, a building layer is another system that uses the given system as input data if the two systems were to be executed in a building process. The functions of the development layers are indicated between columns. Each row indicates the translation process from system conception through actual machine implementation. Each element of the row is data such that each column element of a row is the output data of a translation function for which the input data is the most immediate previous column (e.g., P_HOL is output of the Guide-to-Design function; P_AXES is the input to Guide-to-Design function).

The Guide-to-Design can complement different HOLs. For example, P_HOL_M could be written in DoD1*. P_HOL_S can be written in a test input language (TIL). S is the integrated system of mission and environment subsystems and therefore an HOL program for S is the simulation test system for the target subsystem. The verification functions of the translation tools are indicated within each translation step. The design tools are incorporated in the name of the row element for each translation process.

*DoD Higher Order Language Working Group, "TINMAN," version of "Requirements for Higher Order Computer Programming Languages," March 1976.

TABLE 5.5.1

System Building Matrix Used by Translation Project
Management Personnel to Track Translation Development

DEVELOPMENT LAYERS →

Building LAYERS ↑	DEVELOPMENT LAYERS →				
	ANALYZER ↔	GUIDE TO DESIGN** ↔	COMPILER ↔	ASSEMBLER ↔	BUILD OR ACQUIRE ↔
X	P_AXES _X ACS_AXES _X	P_HOL _X ACS_HOL _X	P_ASS _X	P_MACH _X (TARGET)	
S	P_AXES _S ACS_AXES _S	P_HOL _S ACS_HOL _S (TIL)	P_ASS _S	P_MACH _S (TARGET)	*
M	P_AXES _M ACS_AXES _M	P_HOL _M ACS_HOL _M	P_ASS _M	P_MACH _M (HOST)	

*Could also be hardware such as sensors, I/O devices, etc.

**If resource allocation is used instead of guide to design, ACS_HOL is not needed.

Key To Symbols Used

- X - data with respect to system S
- S - system comprised of mission and environment subsystems (data with respect to M)
- M - management information system

- P_AXES - program written in AXES specification language
- P_HOL - program written in higher order language
- P_ASS - program written in assembly language
- P_MACH - program written in machine language
- ACS - assembly control supervisor
- TIL - test input language

NOTE: Compiler and Assembler are used both as verification and translation tools

Where indicated, the personnel management in the form of the assembly control supervisor (ACS) for each translation process is required. The ACS is required whenever the design is a manual process. Each column of the matrix of Table 5.5.1 indicates one development layer which contains all the building layers. Each row is arranged so that each successive row is a function whose input is the system indicated as the most immediate higher row. For example, system S uses data X as input (i.e., (S(X))); the management information system, M, uses system S as input (i.e., M(S(X))).

This building matrix assumes that the tools used to build the system have themselves been developed and 'frozen' before the translation process that uses them has begun. Each translation is a replacement of the previous translation (e.g., the program in the higher order language (HOL) replaces the AXES specification). Each translation is dependent on the previous development layer. Thus, the AXES specification must be 'frozen' before the HOL program is begun, and so on.

Each row in the matrix can be developed and built separately and in parallel since the interfaces among the layers have been specified.

The manager of a building layer such as system S would have a building matrix showing the subsystem building layers that are monitored by the manager of S. Eventually, there is a manager in charge of a building layer that is divided so that the subsystems are either independent of each other or their communication is that of one system using the output data of another system. At this point, this manager would use a level building matrix which would use the levels of that one layer as the columns of the matrix. The row elements of the matrix using the levels as columns would remain as the development layers of the level subsystems (Figure 5.5.1).

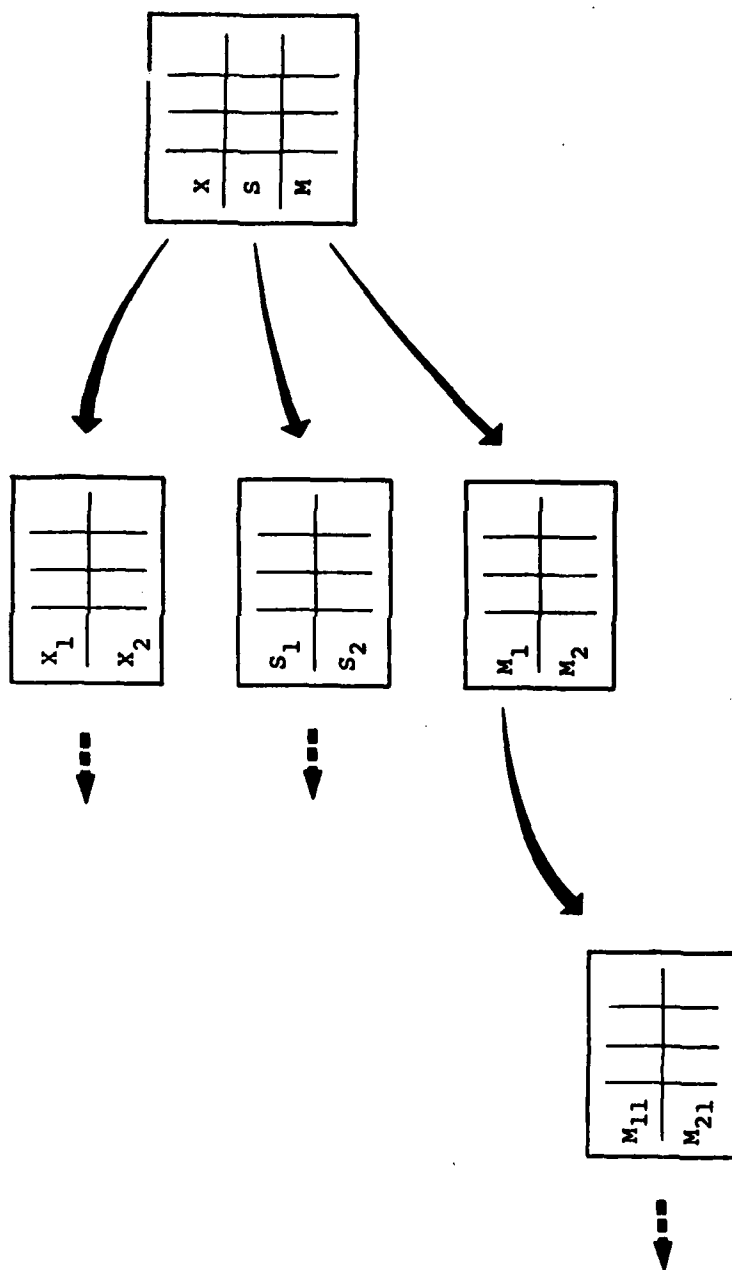


FIGURE 5.5.1: BUILDING MATRICES FOR VARIOUS LEVELS OF SYSTEM DEVELOPMENT

5.5.1 ACS Demonstrated by System Layer Function S

The building process with respect to personnel management and translation process interfaces is shown in Figure 5.5.1.1 for system S. The project manager, the assembly control supervisor (ACS), monitors the task of developing standards to be used as the guide to design:

$$\underbrace{ACS_HOL_S}_{(f_2(P_HOL_S))} = ACS \left(\underbrace{ACS_AXES_S}_{(f_1(P_AXES_S))} \right) \quad (20)$$

The guide to design standards aid in the manual process of using the AXES specification to obtain the HOL resource allocation. The ACS_AXES_S is in charge of producing an AXES specification for system S. This manager manages the official assembly of the specification, P_AXES_S . The ACS_AXES_S monitors the work of managers ACS_AXES_{S1} and ACS_AXES_{S2} . Manager ACS_AXES_{S1} is in charge of producing P_AXES_{S1} , a lower-level AXES specification with respect to P_AXES_S . Likewise, manager ACS_AXES_{S2} is in charge of producing P_AXES_{S2} . Modules P_AXES_{S1} and P_AXES_{S2} must be officially approved by ACS_AXES_S before being allowed into the official assembly. In a similar manner, ACS_AXES_{S1} monitors the activities of the personnel on his next most immediate lower level corresponding to P_AXES_{S1} specifications. This management assignment process can be nested as deeply as desired.

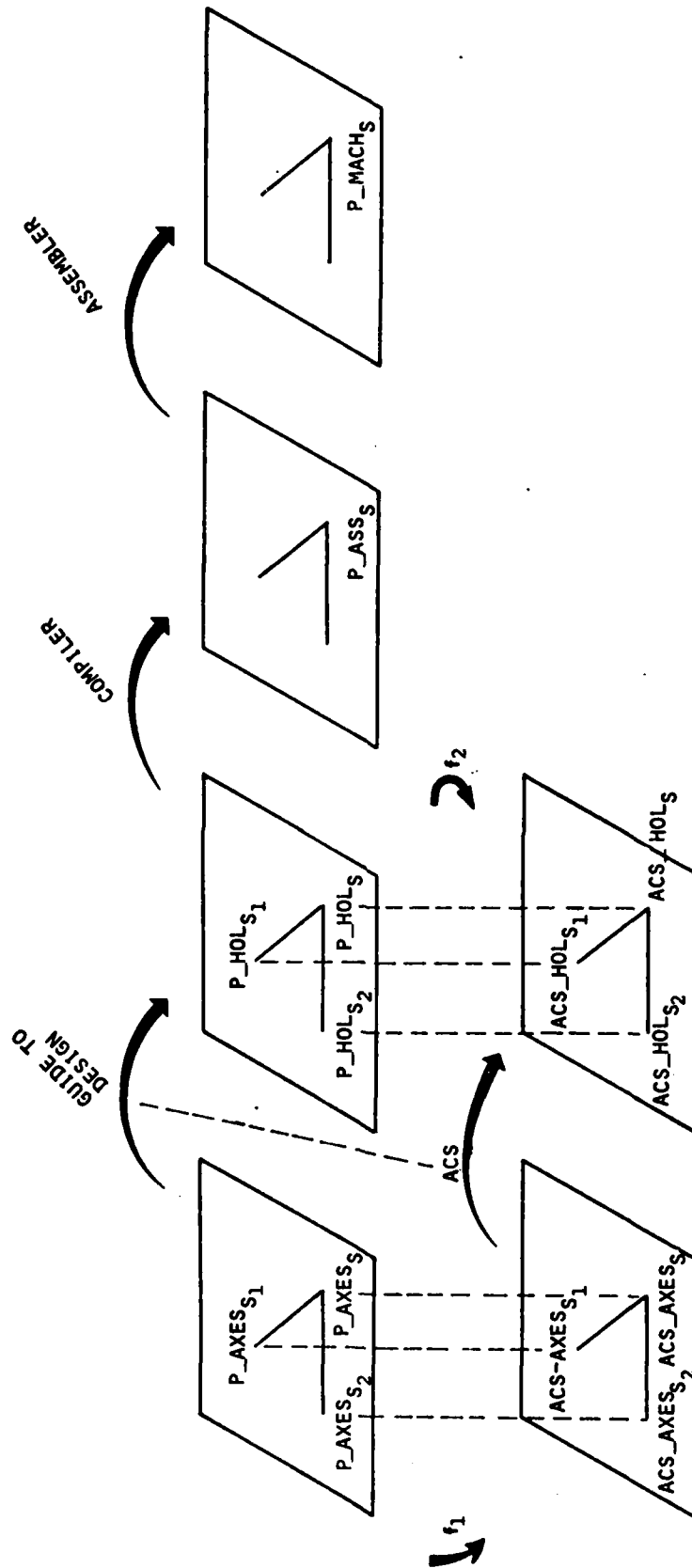


Figure 5.5.1.1: BUILDING PROCESS FOR SYSTEM S WITH RESPECT TO PERSONNEL MANAGEMENT AND THE TRANSLATION PROCESS.

Figure 5.5.1.2 illustrates the ACS concept of managing the module building process. The modules can then be placed into a library to be collected at any time for the present development system or for other systems that need similar capabilities (Figure 5.5.1.3).

The specification analyzer is used by each ACS to verify the interfaces of the functions the ACS is responsible for specifying. Each of these functions can then be developed in parallel. When one of these functions is completely specified (stand alone mode), that function can be verified by the analyzer with the other functions on its own level. The incomplete functions or 'pseudo' modules, can be used with the completed function to verify the interfaces at this time ("with" mode). When the ACS receives all the completed functions that correspond to the level of specification being monitored, the analyzer is used again for verification of the level. In addition, a simulator may be used for performance testing. When this verification process is complete, the level of specification is accepted into the official building assembly for this functional phase of development.

Since levels of P_AXES_S can be developed in parallel, it is possible for a completed specification level to begin the next step in the translation process before the entire P_AXES_S element is complete. ACS_HOL_S monitors the building activity of the personnel assigned to build the HOL program using the same management techniques described for ACS_AXES_S . To verify modules at this translation process we use a compiler. It is possible then, for sections of the HOL program to be built in parallel with sections of the specification process. A library of specification modules is developed in parallel with the development of the library of HOL program modules.

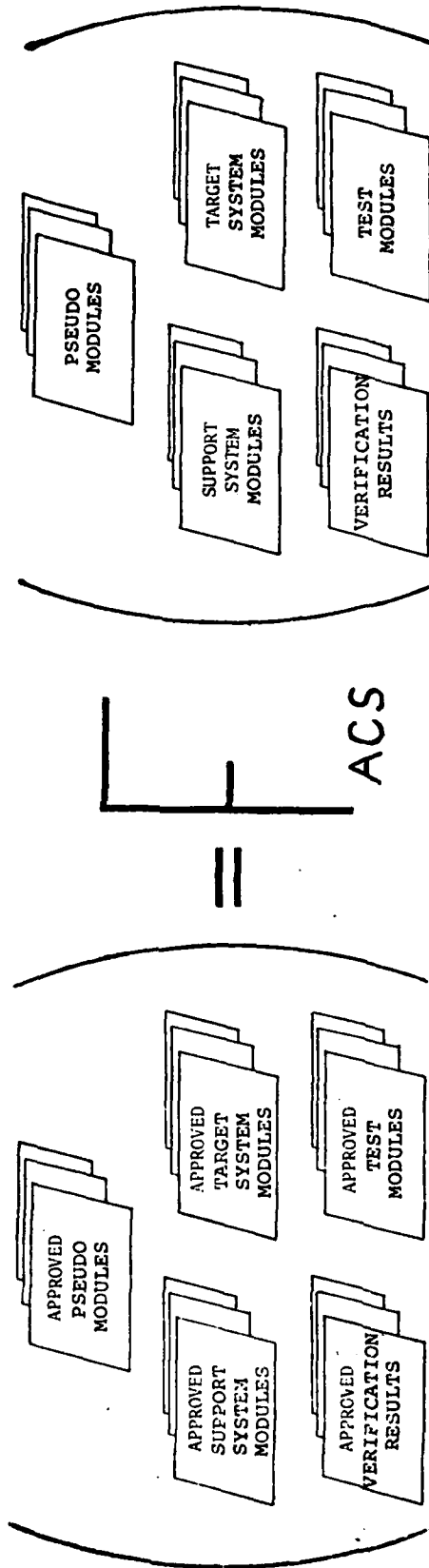


Figure 5.5.1.2: THE ASSEMBLY CONTROL SUPERVISOR CONCEPT

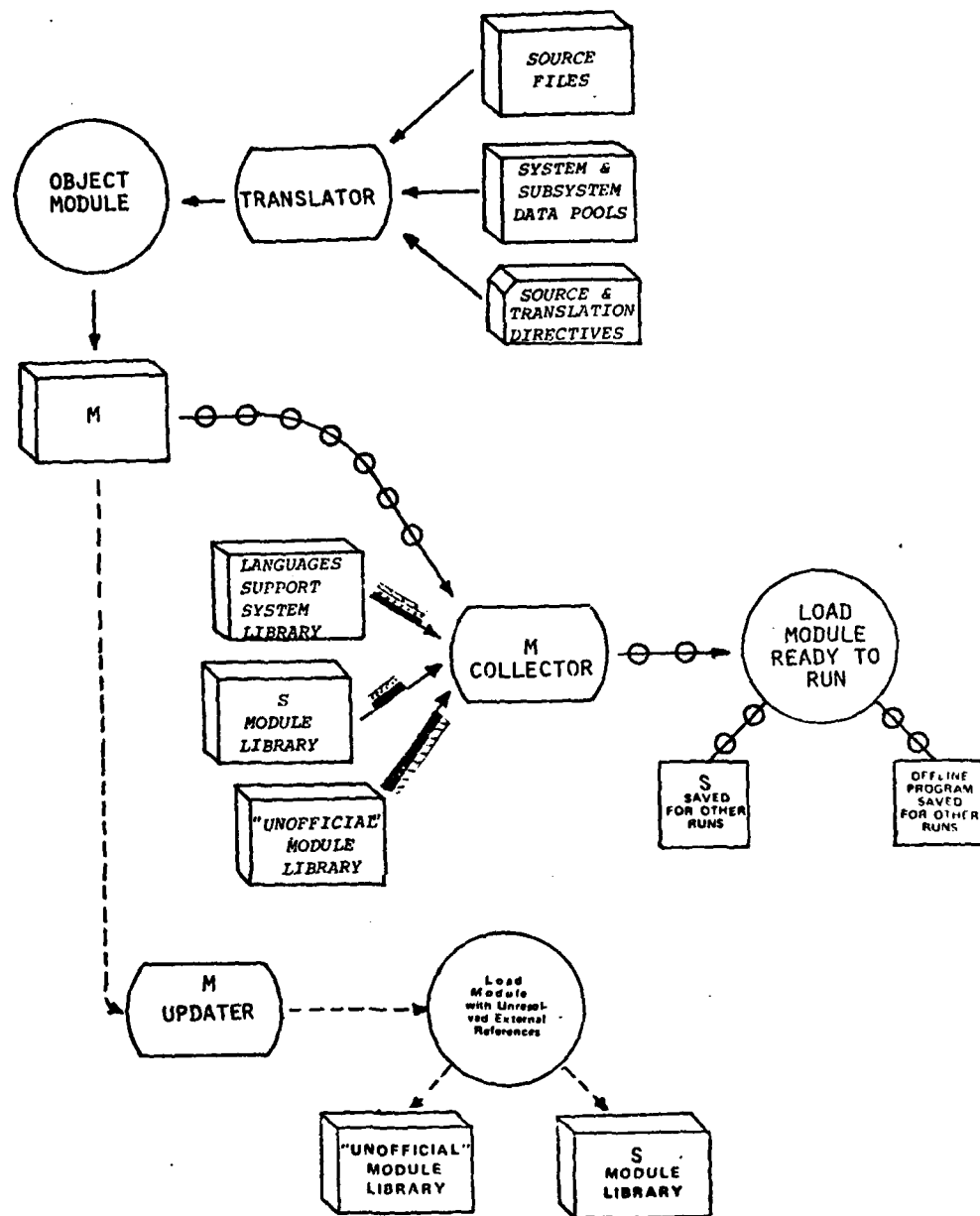
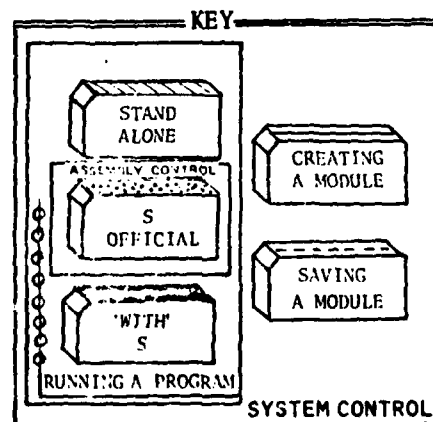


FIGURE 5.5.1.3: BUILDING THE LIBRARY AND THE USE OF THE LIBRARY MANAGED BY ACS



Whenever a modification to an element of a library is made, that change is reflected throughout the translation process by tracking the interfaces via management system, M.

If the translation process from P_HOL_S to P_ASS_S to P_MACH_S is automated by a compiler, there is no need for assembly control supervisor personnel to monitor these activities. In this case, ACS_HOL_S is in charge of the activities from HOL code to machine code. If an automatic tool such as the resource allocator is available, the role of ACS_HOL could also disappear.

Each row of the system building matrix is built in a similar manner as shown for system S. Another example might show the building process for a system in which the translation process to assembly language is a manual process. Here, ACS management would be required to assure that the HOL, implemented in assembly language macros corresponds to the HOL specification. In this case, the guide to design function would again aid the verification, this time from HOL program to the assembly language code.

5.5.2 The Management Building Layer (M)

Table 5.5.1 assumes the use of an automated management system, M, to aid in the building process. System M uses $S(X)$ as input in order to collect statistical information about the elements of S, to relate the development status of the elements of S; and to modify the elements of S.

In the official building process for the entire system, M, it is recommended that some implementations of M be 'frozen' first so that M can collect data for the other systems being developed. The M system itself is a collection of library modules. The interfaces of system M with respect to system S is shown in Figure 5.5.2.1. System function M uses data S(X) (in any form, e.g., P_AXES_S, P_HOL_S) and produces a data base. The personnel management (ACS) use M to record statistical information each time a change is made to the functions the ACS is monitoring. System M will provide standard ways for recording the information that must be incorporated manually (e.g., program change request (PCR) numbers, reason for new revision) and can automatically collect information that in past development efforts either have never been recorded or have been recorded manually (i.e., where error was found, what tool found the error, average execution time for a module, etc.).

System M is also used to build libraries of system functions and to correlate this information with the corresponding function. For example, M can 1) collect P_AXES_{S1} with P_AXES_{X1} and P_AXES_{S2} with P_AXES_{X2} to form P_AXES_S with P_AXES_X, or collect P_HOL_{S1} with P_HOL_{X1}, and P_HOL_{S2} with P_HOL_{X2} to form P_HOL_S with P_HOL_X; 2) associate the documentation for an element in Table 5.5.1 with its corresponding function revision by revision (e.g., a control map for P_AXES_S; a design diagram for P_HOL_S; a report form for ACS_AXES_S).

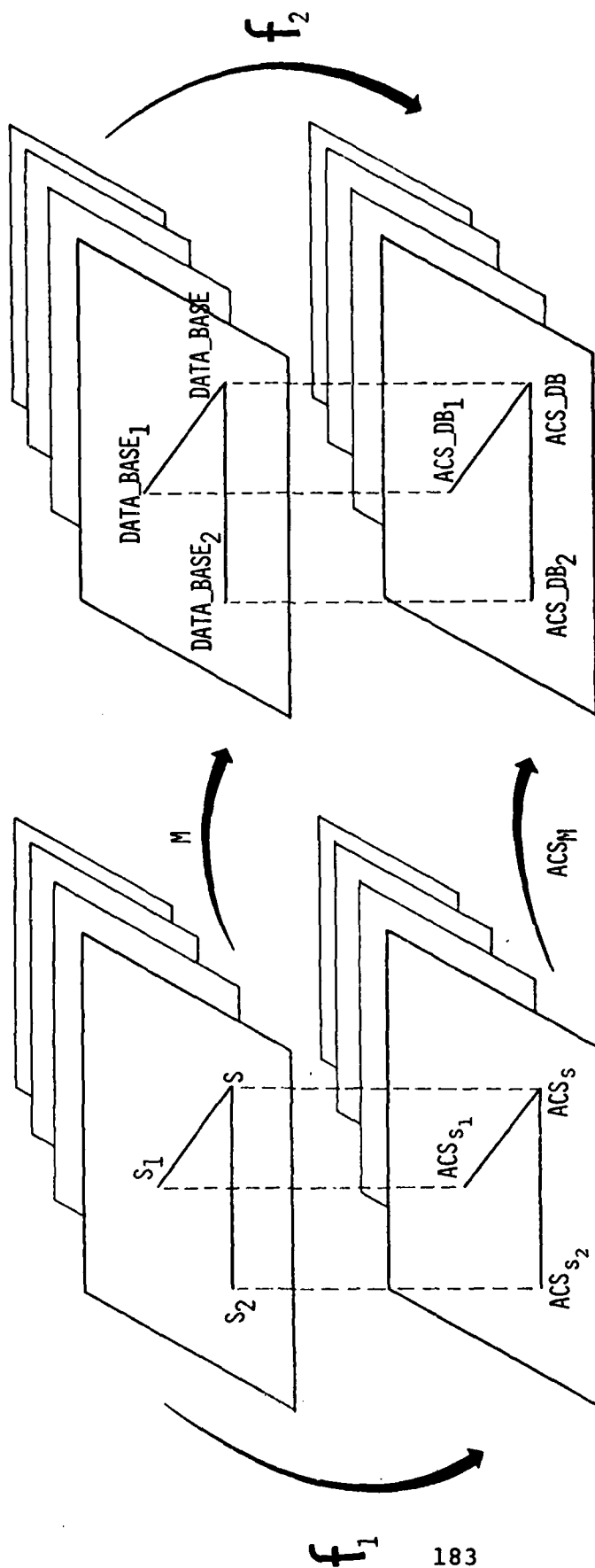


Figure 5.5.2.1.1: SYSTEM LAYER FUNCTION M WITH RESPECT TO SYSTEM S AND THE PERSONNEL MANAGEMENT

Layer M has its own personnel management layers (Figure 5.5.2.1) and its own translation project management process (Table 5.5.1). The subsystems of layer M interface with the subsystems of S in a similar manner. ACS_M (Figure 5.5.2.1) monitors the task of building the M layer function whereas ACS_{DB} monitors the task of building the data base:

$$\underbrace{(f_2(M(S)))}_{\substack{\text{Data} \\ \text{Base}}} = \underbrace{ACS_M}_{ACS_S}(\underbrace{f_1(S(X))}_{ACS_S}) \quad (21)$$

5.5.3 Subsystems of System S

System S can be divided into two main functions: the environment and the mission functions. Table 5.5.3.1 shows the building matrix for the subsystems of S.

Table 5.5.3.1 is arranged in a manner similar to Table 5.5.1. Here, the application subsystem, S_A , uses data X as input; the environment subsystem, S_E , uses system S_A as input. Just as system S can be built independently from and in parallel with system M, subsystem S_A can be built independently from and in parallel with subsystem S_E . The same translation tools are used to build S_A and S_E that were used to build system S and system M. Again, the translation data is dependent on the previous translation data. The more of this process that can be automated, the more reliable the translation process will become.

TABLE 5.5.3.1

System Building Matrix for Subsystems of S

BUILDING LAYERS ↑	DEVELOPMENT LAYERS ↑				
	ANALYZER	GUIDE TO DESIGN	COMPILER	ASSEMBLER	BUILD OR ACQUIRE ASSEMBLER
X	P_AXES _X ACS _X	P_HOL _X ACS _X	P_ASS _X	P_MACH _X	
S _A	P_AXES _{S_A} ACS _{S_A}	P_HOL _{S_A} ACS _{S_A}	P_ASS _{S_A}	P_MACH _{S_A}	
S _E	P_AXES _{S_E} ACS _{S_E}	P_HOL _{S_E} ACS _{S_E}	P_ASS _{S_E}	P_MACH _{S_E}	

Key to Symbols Used

- X - data for system S_A
- S_A - application system
- S_E - environment system
- P_AXES - specification in AXES
- P_HOL - program written in higher order language
- P_ASS - program written in assembly language
- P_MACH - program written in machine language

We assume in Table 5.5.3.1 that only the AXES and HOL translation steps are managed by assembly control supervisors, since these are manual steps for this building matrix. Management system M can operate on S_A and S_E to collect modules during development and to create system libraries for S_A and S_E .

5.5.4 The Environment Layer Subsystem

Table 5.5.4.1 shows the building matrix for the environment subsystem (S_E) of S. Table 5.5.4.1 is arranged in a manner similar to Table 5.5.1 and Table 5.5.3.1. Here, the environment E function (in the form of models or hardware) uses data XE; the operating system function (OS_E) uses system E as its input data. Each system layer function can, again, be built independently. This process is similar to the process for the integrated system shown in Table 5.5.1 and to the sub-integrated system of Table 5.5.3.1. To build each system layer function, we use the same tools for the translation steps that were used to build the integrated system of Table 5.5.1 and Table 5.5.3.1.

The translation building process for environment, E, modules might be more varied. Some E modules might translate directly from the AXES specification to the hardware; others might be modelled in software and also built in hardware in parallel with the modelling effort; others might only be modelled in software. In Figure 5.5.4.1, a sample system E has five functions on its most immediate lower level in the AXES specification: the universe function, a human operator function, a radar system function, the engine function and the computer function.

To build system E, all five E functions are specified in AXES and monitored by the ACS_E personnel.

TABLE 5.5.4.1

Building Matrix for System S_E , The
Environment System Layers.

BUILDING LAYERS	DEVELOPMENT LAYERS			
	ANALYZER	GUIDE TO DESIGN	COMPILER	ASSEMBLER
X_E	$P_AXES_{XE_ACS_XE}$	$P_HOL_{XE_ACS_HOL_XE}$	P_ASS_{XE}	P_MACH_{XE}
E	$P_AXES_{E_ACS_E}$	P_HOL_E	P_ASS_E	P_MACH_E
OS_E	$P_AXES_{OS_E_ACS_E}$	$P_HOL_{OS_E_ACS_HOL_{OS_E}}$	$P_ASS_{OS_E}$	$P_MACH_{OS_E}$

S_E

187

Key to Symbols Used

- | | |
|---|--|
| X_E - data for System E | P_AXES - program written in AXES specification language |
| E - environment system | P_HOL - program written in higher order language |
| OS_E - operating system for System E | P_ASS - program written in assembly language |
| | P_MACH - program written in machine language |

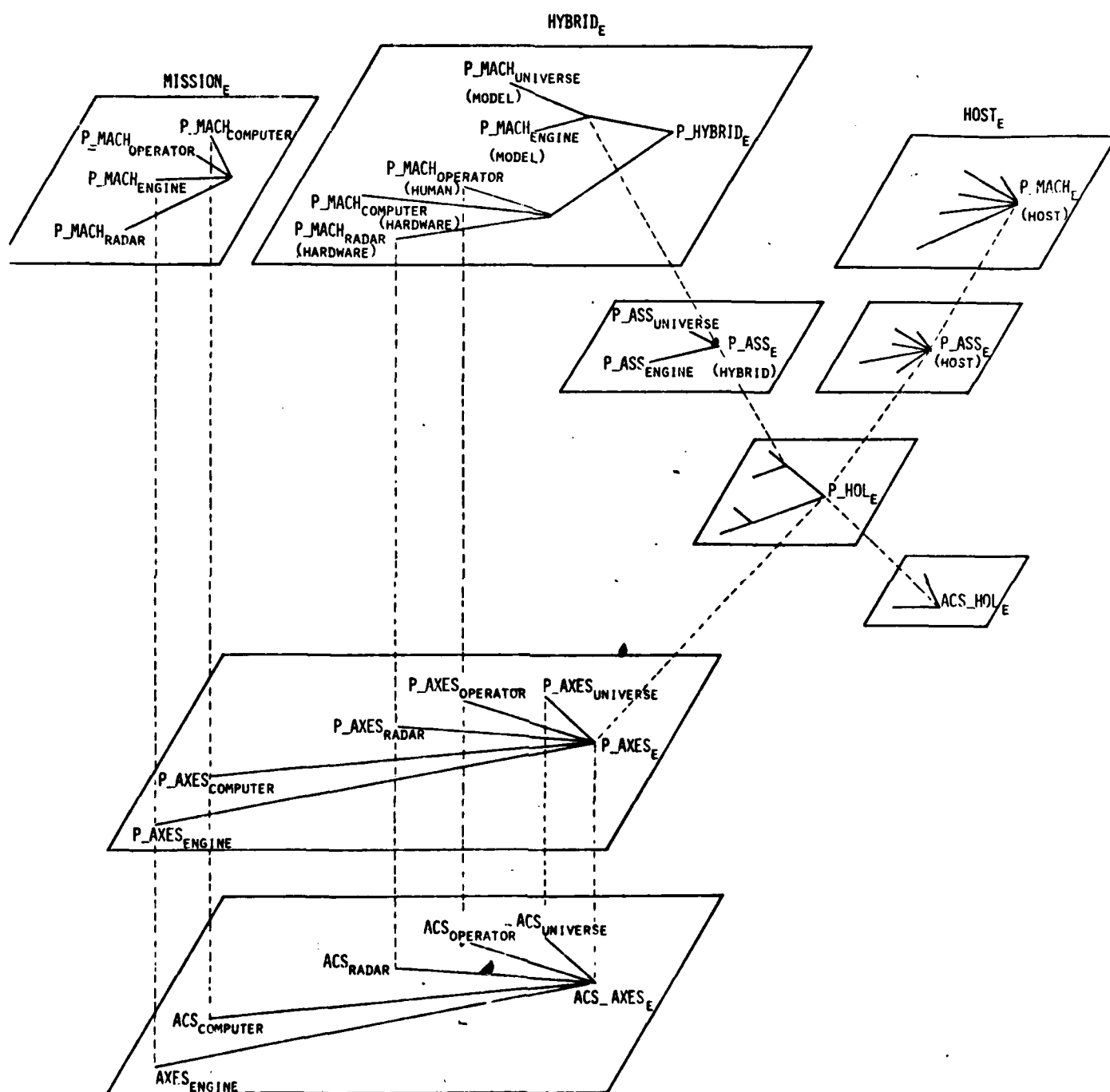


Figure 5.5.4.1: SAMPLE BUILDING PROCESS FOR LAYER SYSTEM FUNCTION E

Once verified and placed in the specification library, the interfaces are determined. These specifications are to be used for an actual mission system, a hybrid computer simulation of the mission and a host digital computer simulation system. The radar is translated directly to hardware and tested on the hybrid machine system before transferring the hardware to the mission software. Here the radar hardware is a 'frozen module' collected in the mission system, i.e., the mission system uses the radar as though it were a library function in that once built for the hybrid, the same radar can be used for the mission.

In parallel with this effort, a model of the radar system is translated to a higher order language and then to the host machine. This modelling effort in the HOL for use on the host machine is done in order to run system S as an integrated system for performance testing the S_A system functions.

The operator function is performed by a person to guide the hybrid simulations for performance testing. The same specifications of the operator are used during the mission. The operator function is modelled in an HOL and translated to the host machine. The HOL for the operator function can be a test input language.

The engine function is built directly from the AXES specifications. Both the universe and engine functions are modelled for the host and hybrid machines. Since S_E uses S_A as input, S_E has the characteristics of the application system so that the computer specification can be developed independent of the application system itself. Once the computer is specified, it can be built directly and also modelled on the host machine. The hybrid system could use the computer hard-

ware as one of its subsystems. During the HOL translation process, the AXES specifications for universe and engine are collected as a unit (AXES function specifications can be regrouped) and compiled. Thus, the universe and engine unit can be translated to both the assembly language of the hybrid and the assembly language of the host without any additional work on the part of the people translating AXES specifications to HOL allocations. The ACS of the HOL translation process assures that both the hybrid version of the universe and engine unit and the host version of that unit are produced.

The hybrid and host machines are used only for performance verification. If an error occurs during a hybrid simulation by either the operator or radar functions, the error can be traced directly to the AXES specifications (Figure 5.5.4.1). If the error occurs in the universe/engine unit, the HOL program is checked first. If no error is found, the program unit in question is traced back to the AXES specification for that unit (Figure 5.5.4.1).

System S_E for the mission is regrouped and collected so that the interface with the real world (i.e., the real universe) is separated from the remainder of system S_E functions.

The environment system S_E is often referred to as a tool with respect to the application system, S_A . When referred to as a tool, S_E is called a simulator.

5.5.5 The Application System Layer S_A

The application system S_A is built as seen in Table 5.5.5.1. Table 5.5.5.1, arranged in a similar manner as the other building matrices, shows A using X_A as input data and OS_A using A as input data. S_A is built and managed in a manner similar to that described for each of the other building processes. Once all the subsystems of S_A are built and verified, S_A as a unit can be placed in the mission system without concern as to its interface consistency with other system S modules.

In Figure 5.5.5.1, function f_1 shows a possible management scheme for S_A functions which is not one-to-one. Here ACS_AXES_A monitors several tasks to be built in the AXES specification.

As soon as the AXES specification is built, some functions of S_A could be built directly in the hardware (Figure 5.5.5.1). For example, we could build a square root function, a matrix data type or even a navigation function directly in the hardware. The resource allocator tool would be very beneficial here, in order to determine the number and type of processors that would best suit the application. If the AXES specification is translated to an HOL, we can use a compiler to build object code for various host machines. This might be necessary if more than one organization of people are involved in building the application system or if one organization of people is responsible for the application system but another organization of people is responsible for the integrated system, S.

TABLE 5.5.5.1

Development Layers →

Building Layers ↑	Development Layers →			
	ANALYZER	GUIDE TO DESIGN	COMPILER	ASSEMBLER
	COMPILER	ASSEMBLER	BUILD OR ACQUIRE	
X_A	$P_AXES_{XA}ACS_{XA}$	$P_HOL_{XA}ACS_{XA}$	P_ASS_{XA}	P_MACH_{XA}
A	$P_AXES_AACS_A$	$P_HOL_AACS_A$	P_ASS_A	P_MACH_A
OS_A	$P_AXES_{OS_A}ACS_{OS_A}$	$P_HOL_{OS_A}ACS_{OS_A}$	$P_ASS_{OS_A}$	$P_MACH_{OS_A}$

S_A

192

Key To Symbols Used

- X_A - data input to A
- A - application system
- OS_A - operating system for A

- P_AXES - specification in AXES
- P_HOL - program written in higher order language
- P_ASS - program written in assembly language
- P_MACH - program written in machine language
- ACS - assembly control supervisor

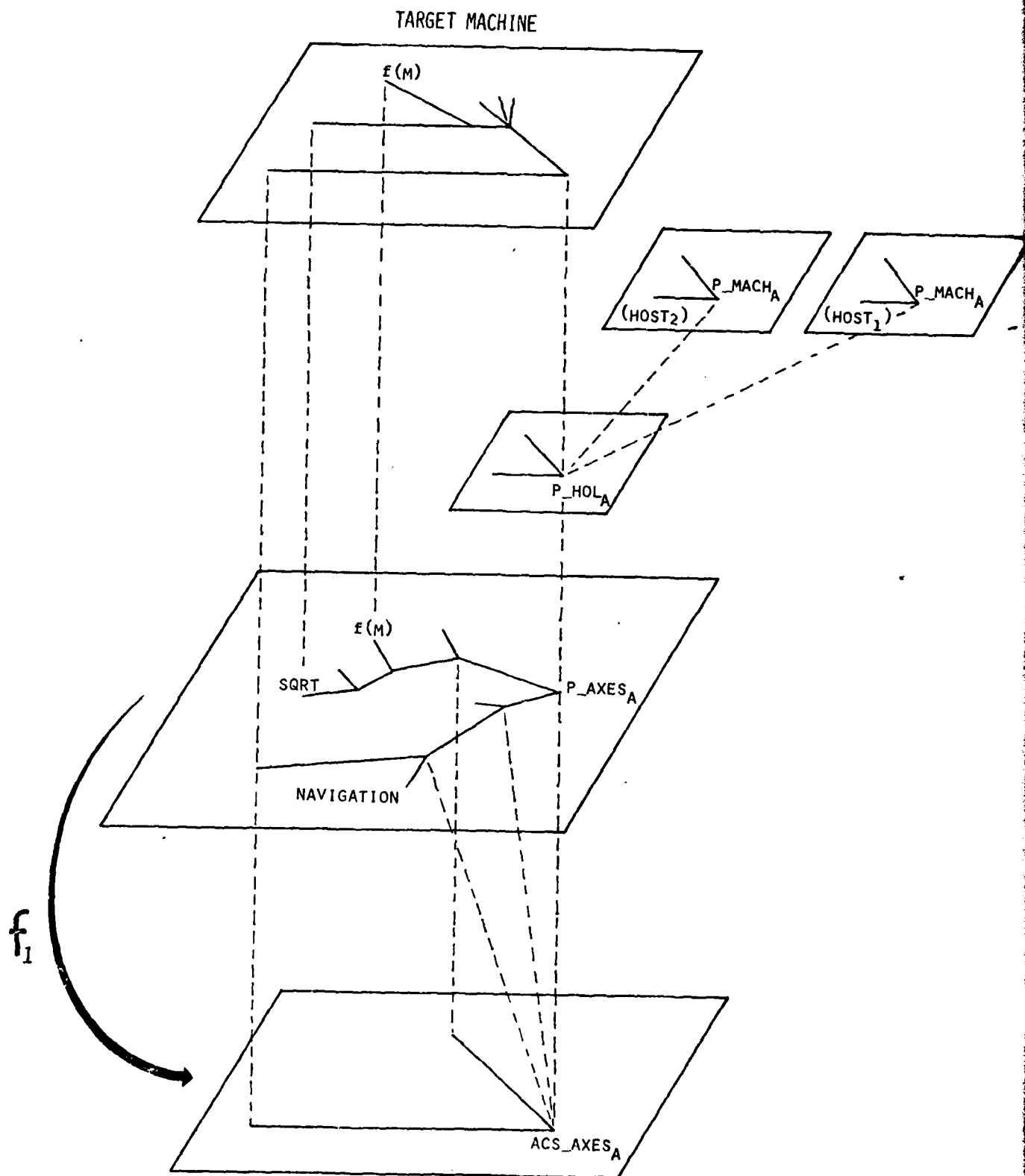


Figure 5.5.5.1: BUILDING THE APPLICATIONS SYSTEM

5.5.6 The Building Levels of Support Tools Functions

The system of support tools used for the translation process can be built and managed in a manner similar to building any other system. Table 5.5.6.1 illustrates the building matrix associated with this building process.

The analyzer uses a specification in AXES, P_AXES, as input. The resource allocator uses the analyzer system output as input. The compiler uses output from the resource allocator as input. These tools use a 'bootstrap' process to get started. For example, the analyzer is written in AXES and then the P_AXES_{AN} is analyzed by the analyzer itself.

5.5.7 'Frozen' Modules

When we divide a system into layers, it becomes apparent which tools, system functions, and system data must be built and the order in which these modules must be built. Figure 5.5.7.1 illustrates the ISDS/HOS building process. AXES and the analyzer must be built and 'frozen' before any other system. Once AXES and the analyzer are 'frozen', system M can be specified in AXES; and HOL, compiler, assembler, and machine can be specified and built in parallel. Once the compiler, assembler and machine are 'frozen', system M modules, specified in AXES, can be built. The development of system S needs the management system to keep track of its building process. Thus, M should be 'frozen' before S is built. System S now has all the support tools necessary for its development. The ISDS/HOS support system can be used over and over again to build any application system other than S.

TABLE 5.5.6.1

Building Matrix for ISDS/HOS Support Tools

Building Levels ↑	DEVELOPMENT LAYERS →			
	P_AXES _{AN}	P_HOL _{AN}	P_ASS _{AN}	P_MACH _{AN}
Analyzer (AN)				
Resource Allocation (RA)	P_AXES _{RA}	P_HOL _{RA}	P_ASS _{RA}	P_MACH _{RA}
Compiler (C)	P_AXES _C	P_HOL _C	P_ASS _C	P_MACH _C
Assembler (AS)	P_AXES _{AS}	P_HOL _{AS}	P_ASS _{AS}	P_MACH _{AS}

Key to Symbols Used

P_AXES	- data for analyzer	P_AXES	- program written in AXES specification language
Analyzer	- translate to control map	P_HOL	- program written in higher order language
Resource Allocation	- translate to HOL language	P_ASS	- program written in assembly language
Compiler	- translate to assembly language	P_MACH	- program written in machine language
Assembler	- translate to machine language		

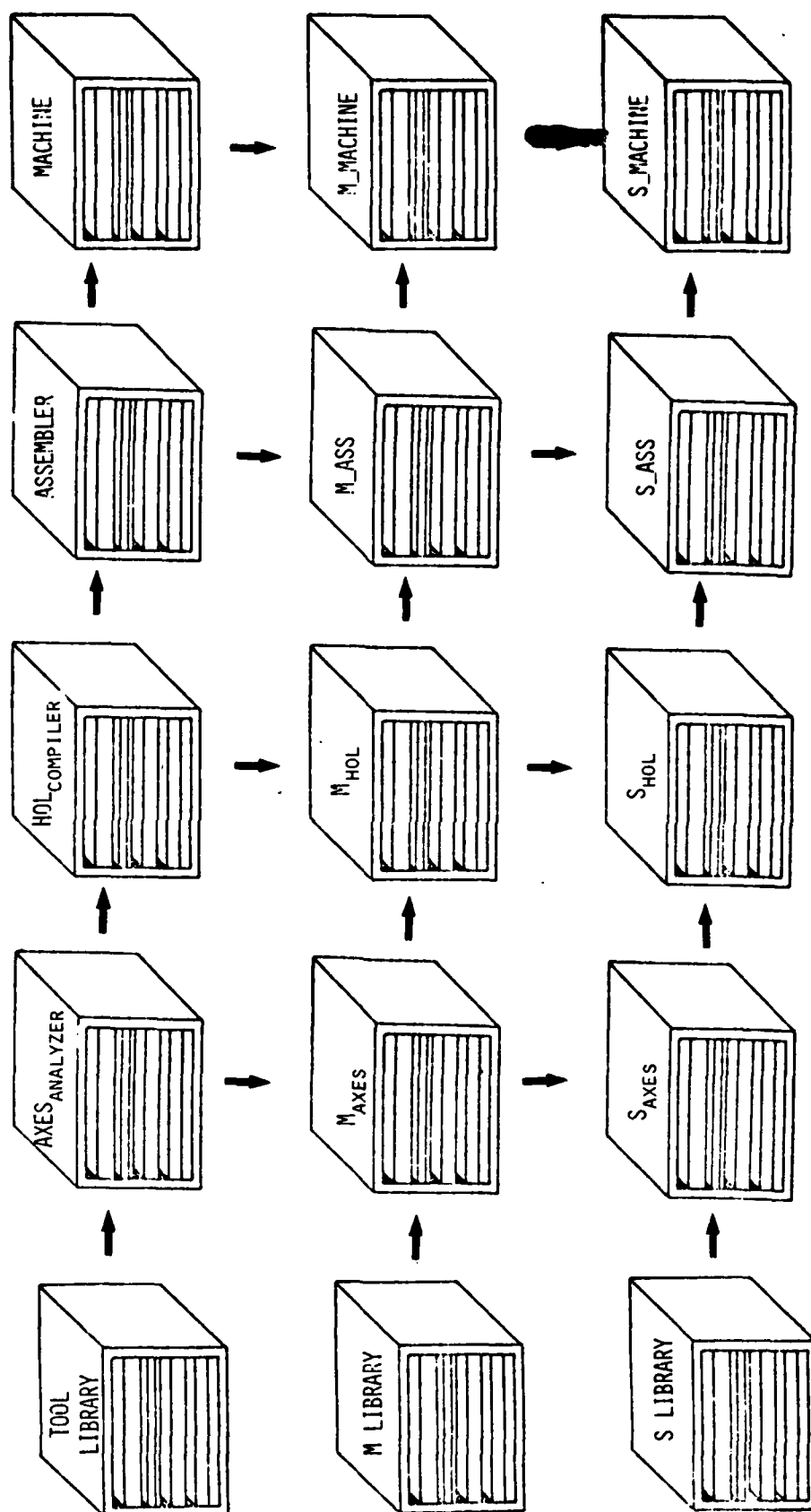


Figure 5.5.7.1: ISDS/HOS Building Process

6.0 TOOLS FOR ISDS/HOS

6.0 TOOLS FOR ISDS/HOS

The previous chapter described the use of ISDS/HOS throughout the life-cycle of computer-based military systems. This chapter presents the automated tools used in the disciplines of design, implementation, documentation, verification, and management of computer-based systems. This chapter is divided into three sections as follows: The first section, Component Tools of ISDS/HOS, describes the associated tools which would enable one to proceed automatically from initial requirements to the target-machine coded form. The second section, Support Tools, describes tools which provide support to project management, interactive development and engineering analysis. The last section, Incremental Tools for Current Use of ISDS/HOS, describes guidelines and conceptual-design modifications to currently available technology, thus providing an incremental approach toward meeting the objectives and concepts of ISDS/HOS. Within each of these sections the conceptual descriptions and the use of the tools as well as their availability will be presented.

6.1 Component Tools of ISDS/HOS

6.1.1 Specification Language (AXES)*

The specification of systems in general and of software systems in particular, has always been a difficult task. This problem increases as systems become more complex. Systems that include

* Excerpted from HAM76b. The specification language AXES is currently being designed and developed at Higher Order Software, Inc. sponsored by the Naval Electronics Laboratory Center.

software as a major component are especially difficult to specify. This is largely due to the fact that software systems are becoming increasingly more complex as a result of advanced hardware technology. The result is that large software systems are often error prone and are always very expensive to develop. Unfortunately, the errors are usually found too late, and the cost of developing systems can far exceed the original estimates. Such system-development processes are usually fragmented. We concern ourselves, therefore, with the beginning of a given system-development process: the specification and its relationship to the total system-development process.

Table 6.1.1.1 lists characteristics of proper specifications. In addition to these characteristics, the specification language should be flexible enough to provide managers of a given application the means to provide standards that satisfy their individual needs. Yet even with this flexibility, the specification languages should also be able to provide the means to communicate any given application system to managers or users of other systems.

A proper specification should be:

- free of errors
- flexible to change
- traceable with respect to a given change
- transferable from one machine to another
- adaptive to different and changing implementations
- easy to define
- easy to use
- easy to understand
- used as either a direct or an indirect means for implementation
- used to maintain proper interfaces throughout a given system development (thus providing for automatic, static verification of interfaces)
- in sufficient and varying degrees of detail so as to satisfy the needs of the
 - manager
 - user
 - systems designer
- able to provide the means of predicting potential problems that would occur in a given implementation

Table 6.1.1.1
Characteristics of Proper Specifications

A successful specification must be designed independent of implicit assumptions; it must be designed independent of its implementation tools; it must be designed independent of system implementation-design concepts; it must convey intent unambiguously; and it must provide mechanisms to describe the properties of systems in as abstract a manner as desired.

A successful specification language should 1) convey to the manager the intent of the specification in a natural manner; 2) provide to the user mechanisms which are easy to use and easy to understand; and 3) provide flexibility to the designer to define any building blocks and be able to show the designer all properties of a given system.

The specification language AXES is intended to provide the mechanisms to define computable systems. These systems include those which are real-time, multi-programmed, or multi-processed. Each system can incorporate built-in error detection and recovery. Towards this end, we are basing the foundation of AXES on a formal methodology which defines a valid specification to be one which is based on completeness of control. The foundations of AXES are based on the methodology of ISDS/HOS. This means that AXES, the building blocks of AXES, and systems defined by AXES will be consistent with the properties of ISDS/HOS.

AXES will provide a reliable means by which to define a successful specification. Reliability is to be obtained from the properties of systems defined by ISDS/HOS. These properties are embedded in the specification techniques for defining abstract control structures. Each abstract control structure uses abstract data types to complete a given system specification.

With AXES, the abstract control structures can be defined in such a way to conform to the formalism required for reliability and to convey intent of the designer to managers and users of a given system application.

It is envisioned that those involved in a system-specification process will have available an option to provide, selectively (and automatically), any layer* of specification. Specifications can be defined in an English-like manner for the manager. Specifications can be defined in terms of representative control structures for the user. Specifications can be defined in terms of their formal definition for the designer.

It is envisioned that the syntax of AXES will become a 'frozen' module, because it is intended that all systems use the same syntax. AXES will be used to design representative abstract control structures and abstract data types as an aid to the user. A system designer can use AXES to design new abstract data types and new abstract control structures. A manager can use AXES to define system standards.

AXES provides for commonality between systems for they will all use the same formal syntax. Many will use the same standard building blocks. Of most importance, all building blocks and specifications will follow the same axioms. Although users of specific applications will have flexibility to choose different building blocks, these building blocks, when "compiled", will be brought to a common meeting ground with all other users of AXES. Again, the adherence to AXES principles is maintained throughout all layers of abstraction.

The basic philosophy of the abstract-building block concept is to treat both the mechanisms of defining a specification and the modules of a specification as if they were 'instructions'. That is, no abstract control structure has any knowledge of a

* Refer to Chapter 5 for definition of "layer".

higher-level abstract control structure using it. Thus, we do not distinguish, in this sense, between an abstract control structure and a module in that they are both systems. In fact, when abstract control structures are defined with specific data types, they are interchangeable with the modules of a system. Ultimately, it is envisioned that systems can be implemented directly from the system specification. Here, the primitive control structures become the instructions of the "machine" itself.

AXES provides the means to define a layered specification in that a "function-first approach" can be used from layer to layer of implementation. Thus, a flexibility exists for choosing mechanisms and allocating resources for a given application. With this approach, a given specification can be transferred to other implementations and their respective machines.

Criteria for AXES Language Statements

Language statements will provide the mechanisms to define a system specification, an abstract data type specification, and an abstract control structure specification. The semantics will include type-checking mechanisms for data types, data structures and control structures.

A system specification is described in AXES by using

1. a set of primitive data types (supplied and implemented in the language); e.g., integers, real, boolean, and string.
2. a set of primitive control structures (also supplied and implemented in the language):
 - composition
 - class partition
 - set partition

3. abstract control structures (the specifications are supplied by the user or by a representative set of "built-in" specifications supplied by the language designers).
4. abstract data types (the specifications are supplied in a manner similar to the abstract control structures).
5. syntax to describe the relationship between the data types, control structures and names of subsystems that comprise a given system.

Language statements will be available to effect the system specification. In order to describe a system specification, the following types of language statements will be included:

- function name identifiers
- data type identifiers
- function equations - implemented data types will have a standard set of operations. Use of abstract data type equations will be limited to the equality operator.
- class construction - used to establish access variables, domain and range for each module.
- function partitions
- function blocks
- parametric replacement statements
- analyzer directives
- comments

Language statement will be available to effect an abstract data type specification. Such a specification consists of three parts:

1. data type name
2. operations on that data type:
 - the domain and range for each operation on the data type can be specified.
3. the set of axioms that define the algebraic specification for a given data type. Each axiom is represented by a system in which:
 - a) the I/O relationship for the system can be determined without a control structure (i.e. by function equations on the data-type), and

- b) the bottom nodes for each system are the operations on the data-type.

Language statements will be available to effect an abstract control structure specification. Such a specification consists of four parts:

1. name and category*
2. semantics
 - a) the set of relations among systems. This includes relationships with other abstract control structures and identification of data types and data structures.
 - b) a control map. This shows the relationships of a system with respect to a given layer.
3. syntax. This considers an English-like equivalent that can be used interchangeably with other control structures.

"Built-In" Subsystems, Abstract Data Types, and Abstract Control Structures

The language syntax will provide only the mechanisms to build systems. It will not include actual system specifications. In order for a systems engineer to be able to use the language effectively, specific "built-in" subsystems, abstract data types and abstract control structures should be provided. The high-level system designer has all the flexibility necessary to create new definitions. The engineer has all the flexibility to use the built-in "subroutines" to establish more complex specifications based on representative system definitions.

* Category depends on type of abstract control structure (HAM76b).

For example, "built-in" subsystems might include functions such as sine, cosine, square root, etc.; "built-in" abstract data-types might include stack, matrix, etc.; "built-in" abstract control structures might include schedule, copy, etc.

"Built-in" subsystems should include abstract control structures so that an option will exist for each module to define:

- (1) specifications for error detection and recovery for each function.
- (2) predicted time and predicted memory usage. (Such information is applicable when it is necessary to predict a system's behavior from the point of view of resource allocation.)

The efficiency and therefore the cost of any given system development is directly related to both reliability and clarity. With AXES, the necessary reliability, the necessary means of communication, and the proper definition of standards can be established for a complete system-development process.

It is envisioned that the cost of a given system development will decrease significantly with the use of AXES.

6.1.2 Design Analyzer

The main function of an analyzer is to guarantee that a given ISDS/HOS system is consistent with the axioms. Automatic interface analysis is provided on a static basis (without execution) by the Design Analyzer.

Real-time software systems cannot be exhaustively tested. The intent of the Design Analyzer is to verify exhaustively and statically a given system defined according to the rules of ISDS/HOS. Interface errors in multiprogramming or multiprocessor systems are caused by data or timing conflicts. Given the ISDS/HOS control system, it is possible not only to design a system with a known and small finite number of logical interfaces to verify, but to prevent both data and timing conflicts. Thus with the Design Analyzer, the more expensive methods of simulation and/or dynamic verification can be limited to unit performance testing.

The Design Analyzer checks the consistency of a specification written in AXES language statements. Figure 6.1.2.1 illustrates the top levels of specification for the analyzer. The Design Analyzer produces a control map as a visual aid in determining valid functional relationships. The Design Analyzer also aids in the design process by means of heuristic algorithms to check the data-type specification and proof-of-correctness algorithms to check the contents of a specification. A manner in which these design verification aids are employed in the Design Analyzer is outlined in what follows.

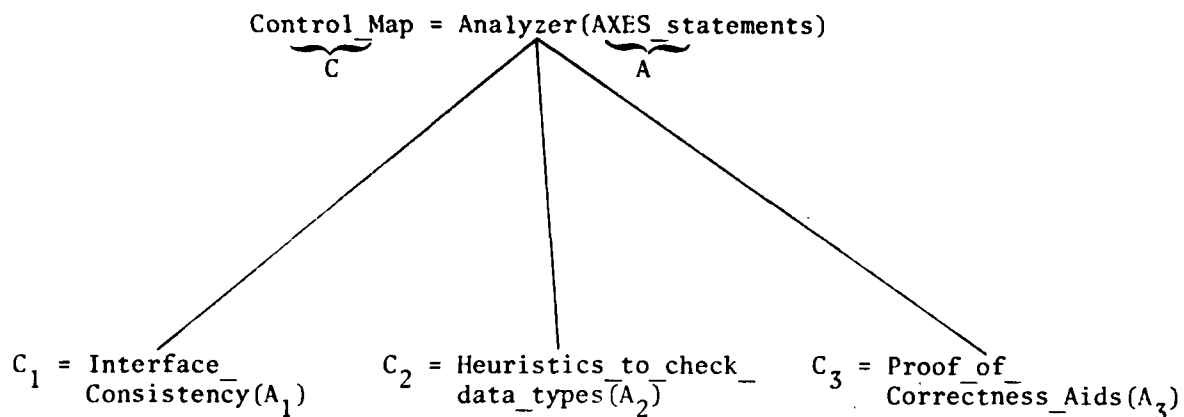


Figure 6.1.2.1

Top Level Decomposition of Analyzer Tool

Interface Consistency

Analysis of a system specification for interface consistency requires a syntactic verification that each function decomposition complies with the axioms. Decompositions that are purely composition, class partition, or set partition can be analyzed directly, according to the considerations outlined for the primitive control structures in Table 5.2.1. These considerations also apply to functional decompositions that have more than two offspring functions but are still of a single type (for example, see Figures 4.3.2 and 4.3.5). Decompositions which combine two or more of the primitive types must first be reconfigured into homogeneous control structures. This can be done automatically by the Design Analyzer through a trace on the input data. If a structure is encountered in the static analysis that cannot be

formed from the primitives, then it is not a legal HOS functional decomposition and will be declared to be in error. All functions which are found to be consistent with the axioms and rules of the structure types would then be guaranteed to interface correctly with the remainder of the system.

Proof-of-Correctness Aids

In program-correctness terminology, a correct program is one for which, given that a specified initial assertion is true preceding a program execution, then its related terminal assertion will be true at program completion. If this definition is translated to ISDS/HOS concepts, a correct specification would be one in which, for each possible input which satisfies an assertion defining the domain of a function, the unique output specified by the function mapping will satisfy a respective assertion defined on the range of the function and the particular input.

To be specific, consider the function,

$$y = F(x)$$

where the variables x and y have been defined as elements of particular data types. For example, a function F , which has input and output assertions, R and P , respectively, could then be

$$Y_{\{y|P(y)\}} = F(X_{\{x|R(x,y)\}}) \quad (1)$$

Thus, for (1), for all x such that $R(x,y)$ is true, the mapping $y=F(x)$ will produce a value of y such that $P(y)$ is also true.

It should be apparent that $R(x,y)$ has explicitly defined the domain of F and $P(y)$ has explicitly defined the range of F . A specific example of such a function description is the square root (SQRT) function.

$$Y_{\{y|y>0\}} = \text{SQRT}(x_{\{x|x=y^2\}}) \quad (2)$$

Here, the relation $x=y^2$ limits the domain of x to positive numbers, as well as describing function SQRT, and the statement $y>0$ limits the range of y to be positive numbers.

Before an examination is made of the implications of correctness assertions on the decomposition of a function, it is appropriate to examine this particular definition of specification "correctness". Assuming that a particular system specification satisfies the preceding criteria of correctness, it still remains to be determined whether the system function actually does the job that is required of it. Obviously, a function which correctly specifies the mapping for assertions of a square-root operation will be of absolutely no value if the operation that is actually desired is the determination of the cosine. Thus it remains the ultimate task of the system designer to assert (i.e., prove) that the correctness assertions defined for the system do actually specify the function called for.

It may be apparent from the correctness assertions themselves that a desirable system must satisfy the specified criteria. If not, then some lower level in the system will be tractable by human understanding and, hopefully, justification of the correctness criteria could be pieced together from that point upward. It is obvious, then, that the determination of proper correctness criteria (i.e., the requirements of the system) is a critical part of the system specification and should be

accomplished as soon as possible in the specification process. However, it may take many iterations through attempted specifications to identify all the deficiencies in the proposed system correctness requirements.

These higher concepts of specification correctness notwithstanding, consider now the implications of functional decomposition on the correctness conditions of a parent function. Because a system control-map specification can always be automatically reconfigured into modules which are primitive control structures, correctness results established for the primitive control structures themselves will generalize to any abstract control structure. The COMPOSITION primitive with correctness assertions could be as shown in Figure 6.1.2.2. This example uses the same form as (1).

$$\begin{array}{c}
 (Y\{Y|P_0(Y)\}) = F_0^X\{X|R_0(X,Y)\} \\
 \swarrow \quad \searrow \\
 (Y\{Y|P_1(Y)\}) = F_1(Z\{Z|R_1(Y,Z)\}) \quad (Z\{Z|P_2(Z)\}) = F_2^X\{X|R_2(X,Z)\}
 \end{array}$$

Figure 6.1.2.2: Assertions for Composition Primitive

The considerations of the input/output assertions in the composition primitive are as follows:

1. obviously, because x is input to both F_0 and F_2 , then $R_0(x,y) \equiv R_2(x,z)$,
2. because the output of F_2 is defined to be the identical value input to F_1 , in the value of z , then $P_2(z) \equiv R_1(y,z)$,
3. because the output of F_1 is identical to the output of F_0 , this being the particular value of y , then y must satisfy both P_0 and P_1 . However, the value y of $P_1(y)$ must also satisfy $R_1(y, \cdot)$. Thus for P_0 to be true, both P_1 and P_2 must also be true for the same values of x , y , and z . Thus, $P_0(y) = P_1(y) \text{ AND } P_2(z)$.

Therefore, the functional decomposition of F_0 into F_1 and F_2 implies that the output condition P_0 of F_0 is also decomposed into the output conditions P_1 of F_1 and P_2 of F_2 .

Similarly, other functions decomposed by other control structures can be described by input/output assertions. The input and output assertions would be supplied by the system designer to aid in system specification as well as substantiate the final product. It may be possible for the Design Analyzer to verify proper decompositions of assertions automatically, according to the characteristics of the primitive control structures. This verification process can be applied at all stages of system decomposition to help the designer define consistent and complete input and output requirements.

Data-Type Analysis

The definition of data objects and their operations is an integral part of a system specification. The data-type specification techniques outlined in Section 6.1.1 for ISDS/HOS are adapted from the algebraic approach of Guttag (GUT75). Using this axiomatic approach, it is possible to define abstract

data types and the primitive operations performed on them without prescribing a manner of implementation. For a data type defined in this manner to be useful, (1) the data type characterized by the axioms must actually be the desired data objects, (2) the axioms must be consistent with one another, and (3) the axioms must comprise a complete set in that they are sufficient to define the meanings of the primitive operations.

The problem of guaranteeing that axioms are consistent is much simpler for the system designer than that of insuring that every necessary characteristic of the desired data object has been included. This fact is reflected in the formal theory of algebraic data-type specification. It has been shown that it is impossible, in general, to verify a set of axioms mechanically. The completeness and consistency of a data-type specification can be mechanically guaranteed, however, if the designer systematically limits the complexity of the axiom definitions and requires that the operations be both primitive recursive and total. These limitations appear not to eliminate any axiomatizations that might be useful in specifying computational systems. The complexity of the axioms may also have to be limited to allow human comprehension of the overall data-type specification. This may be necessary to guarantee that the defined data object is really the desired one.

This systematic procedure can be implemented as automatic design-aid heuristics. These can interactively assist the system designer in generating a set of algebraic data-type axioms that are suitable to mechanical verification of consistency and completeness. As outlined, an abstract data-type specification process would begin with the syntactic definition of the primitive-operation interfaces (see Section 6.1.1). The system designer will derive this information from what he knows about

the desired operations, their domains, and their ranges. Given this, the design-aid tool can query the designer for more specific definition where required, or reject inconsistent specifications, until a complete and consistent set of axioms is defined for the operations. Once the data types have thus been specified, the Design Analyzer can then verify the axiomatic consistency of each instance of a primitive operation in a system control map.

6.1.3 Static Resource Allocation Tool (RAT)

The Resource Allocation Tool will automatically generate machine executable code for a target machine from a system-specification control map. The RAT will provide a vital link in automating the process of software production by eliminating the expense and error of manual allocation of computer resources and computer-program implementation. This tool will generate a software configuration from a control map and optimize this configuration to the particular implementation requirements.

To perform these functions, the Resource Allocation Tool must first reconfigure a system-specification control map into a standard architectural form still independent of hardware considerations. The architectural form presents the system specifications in a communicable format, suitable for use by designers in the selection of hardware. An architectural form is a reduction of the control map into as few levels as possible.

From the architectural form, the RAT can analytically determine both the time-optimal and memory-optimal software configurations. The time-optimal configuration will be found under the assumption that an unconstrained number of multiple processing units, as well as unlimited memory space is available. To determine the memory-optimal configuration, it must be assumed that unlimited execution time is available and that a single processing unit is executing. The time-optimal configuration will make apparent the time-critical execution paths in which data dependencies require sequential execution. The memory-optimal configuration will illuminate the dynamic memory-allocation possibilities for both data and instructions. The time-optimal and memory-optimal configurations will provide bounds on the possible time/memory performance trade-offs that must be made in the determination of an overall optimal configuration for a particular hardware implementation.

The next step in resource allocation is to generate an optimal software-module configuration according to specified implementation constraints. The RAT will accept time and memory constraints and system input-data information and apply analytical optimization techniques and functional-simulation optimizing techniques to arrive at an optimal configuration. The RAT then generates machine code for a particular machine. It will receive the specific machine parameters as input and produce optimal machine executable code as output, entirely bypassing the traditional HOL step of system development. The inputs and outputs of the RAT are given in Table 6.1.3.1.

Resource Allocation With Primitive Control Structures

System specifications can be expressed as a control map containing only primitive control structures. Insight into the resource allocation for abstract control structures can be gained by examining the resource allocation for primitive control structures. All of the concepts discussed below also apply to more complex data types and control structures.

COMPOSITION

Figure 6.1.3.1 illustrates the primitive control structure of composition. Obviously, sequential execution is implied; function f_2 must be performed before f_1 , because f_1 requires the value of z as input. z itself might consist of more than one variable. Consider $z = (u,v)$. In this case, if f_2 were to generate the value of v before that of u , f_1 could begin executing those of its internal functions which require only the value of v as input. When u is subsequently defined by f_2 , the remainder of f_1 could begin executing.

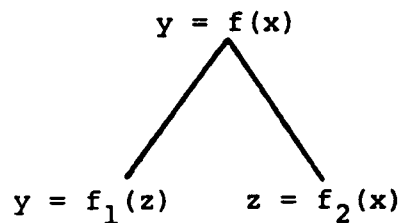


Figure 6.1.3.1

The Primitive Control Structure: Composition

There are additional allocation implications of the composition primitive structure. These are as follows:

1. z exemplifies the generation of data that is local to a module. Storage space for z does not need to be allocated until f_2 has generated its value. When f_1 has completed its references to z , the storage for z may be released.
2. If x has no other references outstanding in modules above f in the control tree, then the storage for x may be released as soon as f_2 completes its references to x .
3. The storage for y need not be allocated until its value is generated by f_1 .
4. Because f_2 is not needed after z is defined, the storage for its instructions may be released when z is defined.
5. Similarly, the storage for f_1 's instructions need not be allocated until z is defined.
6. Because the data dependency of f_2 and f_1 imply a sequential execution, f_1 and f_2 might best be performed within a single processor to minimize the overhead of transfers. However, if it can be assumed that f_1 will be able to begin execution before all of f_2 is completed, it might be advantageous to execute these functions on different processors.

Table 6.1.3.1
Input and Output of the Resource Allocation Tool

INPUT

- A. System-specification control map.
- B. Target-machine configuration parameters.
- C. System-operation input-data distribution statistics (or estimates).

OUTPUT

- A. Standard Architectural form of control map.
- B. Optimal software configuration for general hardware configuration as specified by:
 - (1) maximum number of usable processing units.
 - (2) maximum available storage space.
 - (3) upper bound on permissible execution-time statistics
- C. Optimal machine-executable code for target machine as specified by:
 - (1) mapping of standard abstract-machine instruction set into target-machine instruction set.
 - (2) memory-storage address-space size- and access-constraints.
 - (3) number of data banks.
 - (4) number of data-access ports for each processor
 - (5) number of registers available.
 - (6) amount of cache memory available to each processor.

Set Partition

Figure 6.1.3.2 illustrates the primitive control structure of set partition. Set partition provides the means for a decision process; only one of the functions f_1 or f_2 is performed for any given invocation of the module. Consequently, the decision to execute f_0 or f_1 cannot be made until the value of x has been determined.

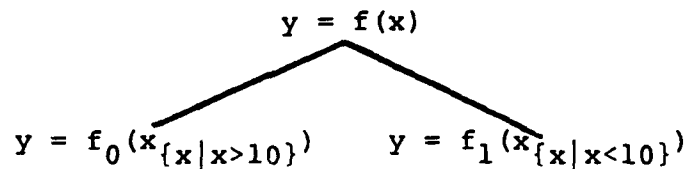


Figure 6.1.3.2
The Primitive Control Structure: Set Partition

Because neither the output of f_0 or f_1 can be produced before the value of x is determined, it might be desirable to postpone loading instructions for f_0 or f_1 until that time. However, it is also possible that the time overhead of loading f_0 or f_1 dynamically may be more "expensive" than loading both f_0 and f_1 into storage before the value of x is determined.

Class Partition

Figure 6.1.3.3 illustrates the primitive control structure of class partition. Functions f_1 and f_2 are strictly independent and may be executed on different processors provided the extra overhead is justified.

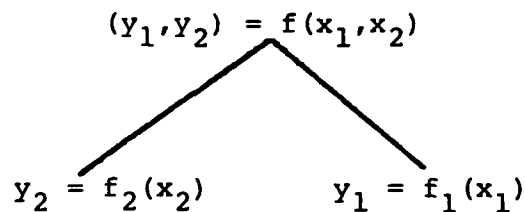
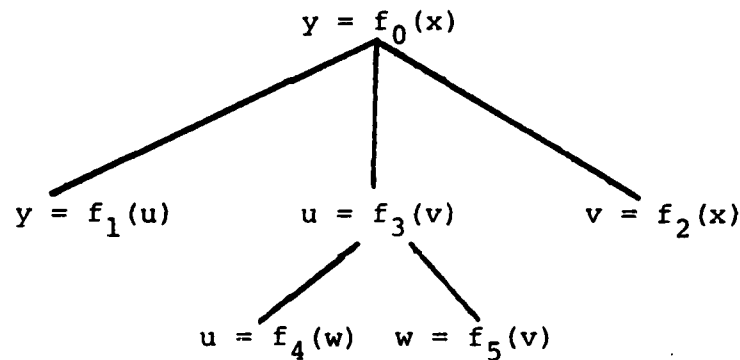


Figure 6.1.3.3

The Primitive Control Structure: Class Partition

Standard Architectural Form

The architectural form of a control map provides a standard means to communicate a specification and provides insights into general hardware requirements. It is a reduction of the control map into as few levels as possible, with the set partition being the only control structure that would not be condensed into its controller. Figures 6.1.3.4 and 6.1.3.5 illustrate the condensing of primitive control structures.



condenses to

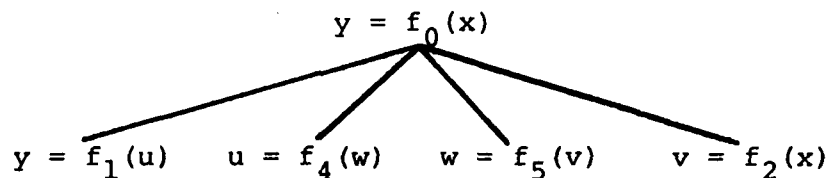


Figure 6.1.3.4
Condensing of the Composition Primitive

The architectural form provides a maximal grouping of data-dependent execution paths. These are the critical paths of execution in which the output of one operation is needed as input to another, this being simply an occurrence of the composition primitive. When a control map is condensed into the architectural form, these paths are carried out as long as possible within a single module. Module invocation in a set partition control structure is the only interruption in the data-dependent paths, because execution flow in a set partition depends on the values of the test data and cannot be determined statically.

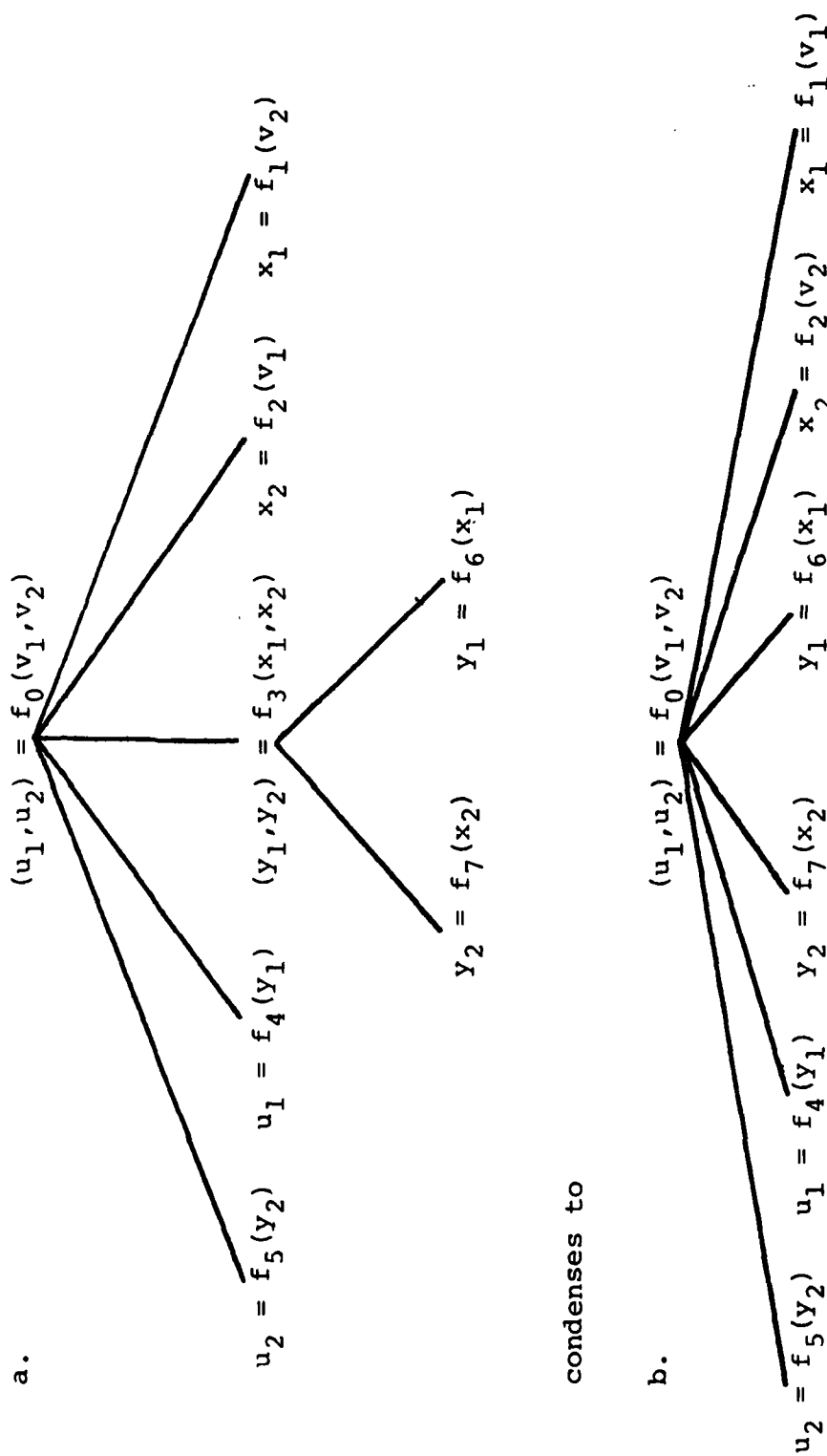


Figure 6.1.3.5
Condensing of the Class Partition Primitive

Optimization Techniques

It is possible to apply both analytical and heuristic optimizing techniques in the RAT. These would be predominantly based on the critical paths of data-dependent execution. In an HOS specification, these paths can be traced explicitly, down to the primitive machine-operation level.

While it is impossible to determine the flow of execution in a system statically, it is possible to generate statistics on the average execution flow through functional simulation of module invocation. From a priori distribution statistics on the system input data (measured or estimated), it is possible for the RAT to determine invocation statistics for each module in the system by means of a top-down analysis of the data distributions at each level. With the invocation-distribution statistics and execution-time estimates for each module in the system, the RAT could use analytical or simulation techniques to estimate system-execution times and resource utilization.

The statistics for the unconstrained time-optimal and memory-optimal configurations will provide bounds within which the RAT can optimize the trade-offs of configurations evaluated by functional simulation.

Code Generation

Once the architectural form of the control map has been produced and the data-dependent paths have been optimized for a given implementation, the RAT would generate the required target-machine code.

The bottom level of the architectural form of the control map contains the primitive machine operations. These primitive operations would be placed in the sequence dictated by the architectural form. Utilizing the dynamic memory allocation and time-critical paths indicated by the optimization techniques,

optimal target-machine code will be produced for a given implementation.

Resource Allocation Example

Figure 6.1.3.6 is an example of possible alternative ISDS/HOS resource allocations for the following expression.

$$x = \frac{(a+b) * (c-d)}{(e-f) * (g+h)}$$

Figure 6.1.3.6(a) depicts the evaluation of the expression through sequential programming. Using this method, memory cannot be allocated dynamically because it is impossible to determine at what point in the execution the storage for a datum may be reallocated. As a consequence, memory for all the data must be allocated before execution.

Figures 6.1.3.6(b), (c), and (d) show three possible alternatives ISDS/HOS resource allocations for the same expression. In each of these alternatives, memory may be allocated dynamically. As prescribed by the axioms of ISDS/HOS, a datum is assigned a value only once and referenced only once. Single reference implies that a new name is used for each reference to a datum. This may be implemented with a single name or address for the variable and a counting mechanism to determine when the storage for that datum may be reallocated. In either case, the number of references to a datum and how they are made is known statically from the specification.

Figure 6.1.3.6(b) shows the time-optimal resource allocation of the expression. Its evaluation requires only three execution steps compared to the seven steps of sequential programming. While the time-optimal decomposition requires

$$x = \frac{(a+b) * (c-d)}{(e-f) * (g+h)}$$

SEQUENTIAL PROGRAMMING

Possible Alternative ISDS/HOS Resource Allocations

ALLOCATE 10 MEMORY											TIME OPTIMAL	MEMORY OPTIMAL	TIME MEMORY CONSTRAINED	TRADE OFF PROCESSORS																																										
X	R ₁	R ₂	a	b	c	d	e	f	g	h																																														
1	2	3	4	5	6	7	8	9	10	11																																														
<p>STEP 1: R₁ = a+b</p> <p>STEP 2: R₂ = c-d</p> <p>STEP 3: R₁ = R₁*R₂</p> <p>STEP 4: R₂ = e-f</p> <p>STEP 5: R₁ = R₁/R₂</p> <p>STEP 6: R₂ = g+h</p> <p>STEP 7: X = R₁/R₂</p>											<p>STEP 1 MEMORY</p> <table border="1"> <tr> <td>t₁ = a+b</td> <td rowspan="4">12</td> </tr> <tr> <td>t₂ = c-d</td> </tr> <tr> <td>t₃ = e-f</td> </tr> <tr> <td>t₄ = g+h</td> </tr> </table> <p>STEP 2 MEMORY</p> <table border="1"> <tr> <td>t₅ = t₁*t₂</td> <td rowspan="2">6</td> </tr> <tr> <td>t₆ = t₃*t₄</td> </tr> </table> <p>STEP 3 MEMORY</p> <table border="1"> <tr> <td>x = t₅/t₆</td> <td>3</td> </tr> </table> <p>AVERAGE MEMORY USE PER STEP: 11</p> <p>EXECUTION TIME: 7 STEPS</p> <p>(a)</p>	t ₁ = a+b	12	t ₂ = c-d	t ₃ = e-f	t ₄ = g+h	t ₅ = t ₁ *t ₂	6	t ₆ = t ₃ *t ₄	x = t ₅ /t ₆	3	<p>STEP 1 MEMORY</p> <table border="1"> <tr> <td>t₁ = a+b</td> <td rowspan="2">9</td> </tr> <tr> <td>c,d,e,f,g,h</td> </tr> </table> <p>STEP 2 MEMORY</p> <table border="1"> <tr> <td>t₂ = c-d</td> <td rowspan="2">8</td> </tr> <tr> <td>t₁,e,f,g,h</td> </tr> </table> <p>STEP 3 MEMORY</p> <table border="1"> <tr> <td>t₅ = t₁*t₂</td> <td rowspan="2">7</td> </tr> <tr> <td>e,f,g,h</td> </tr> </table> <p>STEP 4 MEMORY</p> <table border="1"> <tr> <td>t₃ = e-f</td> <td rowspan="2">6</td> </tr> <tr> <td>t₅,g,h</td> </tr> </table> <p>STEP 5 MEMORY</p> <table border="1"> <tr> <td>t₄ = g+h</td> <td rowspan="2">5</td> </tr> <tr> <td>t₃,t₅</td> </tr> </table> <p>STEP 6 MEMORY</p> <table border="1"> <tr> <td>t₆ = t₃*t₄</td> <td rowspan="2">4</td> </tr> <tr> <td>t₅</td> </tr> </table> <p>STEP 7 MEMORY</p> <table border="1"> <tr> <td>x = t₅/t₆</td> <td>3</td> </tr> </table> <p>AVERAGE MEMORY USE PER STEP: 6</p> <p>EXECUTION TIME: 7 STEPS</p> <p>(c)</p>	t ₁ = a+b	9	c,d,e,f,g,h	t ₂ = c-d	8	t ₁ ,e,f,g,h	t ₅ = t ₁ *t ₂	7	e,f,g,h	t ₃ = e-f	6	t ₅ ,g,h	t ₄ = g+h	5	t ₃ ,t ₅	t ₆ = t ₃ *t ₄	4	t ₅	x = t ₅ /t ₆	3	<p>STEP 1 MEMORY</p> <table border="1"> <tr> <td>t₁ = a+b</td> <td rowspan="3">10</td> </tr> <tr> <td>t₂ = c-d</td> </tr> <tr> <td>e,f,g,h</td> </tr> </table> <p>STEP 2 MEMORY</p> <table border="1"> <tr> <td>t₃ = e-f</td> <td rowspan="3">8</td> </tr> <tr> <td>t₄ = g+h</td> </tr> <tr> <td>t₁,t₂</td> </tr> </table> <p>STEP 3 MEMORY</p> <table border="1"> <tr> <td>t₅ = t₁*t₂</td> <td rowspan="2">6</td> </tr> <tr> <td>t₆ = t₃*t₄</td> </tr> </table> <p>STEP 4 MEMORY</p> <table border="1"> <tr> <td>x = t₅/t₆</td> <td>3</td> </tr> </table> <p>AVERAGE MEMORY USE PER STEP: 6.75</p> <p>EXECUTION TIME: 4 STEPS</p> <p>(d)</p>	t ₁ = a+b	10	t ₂ = c-d	e,f,g,h	t ₃ = e-f	8	t ₄ = g+h	t ₁ ,t ₂	t ₅ = t ₁ *t ₂	6	t ₆ = t ₃ *t ₄	x = t ₅ /t ₆	3
t ₁ = a+b	12																																																							
t ₂ = c-d																																																								
t ₃ = e-f																																																								
t ₄ = g+h																																																								
t ₅ = t ₁ *t ₂	6																																																							
t ₆ = t ₃ *t ₄																																																								
x = t ₅ /t ₆	3																																																							
t ₁ = a+b	9																																																							
c,d,e,f,g,h																																																								
t ₂ = c-d	8																																																							
t ₁ ,e,f,g,h																																																								
t ₅ = t ₁ *t ₂	7																																																							
e,f,g,h																																																								
t ₃ = e-f	6																																																							
t ₅ ,g,h																																																								
t ₄ = g+h	5																																																							
t ₃ ,t ₅																																																								
t ₆ = t ₃ *t ₄	4																																																							
t ₅																																																								
x = t ₅ /t ₆	3																																																							
t ₁ = a+b	10																																																							
t ₂ = c-d																																																								
e,f,g,h																																																								
t ₃ = e-f	8																																																							
t ₄ = g+h																																																								
t ₁ ,t ₂																																																								
t ₅ = t ₁ *t ₂	6																																																							
t ₆ = t ₃ *t ₄																																																								
x = t ₅ /t ₆	3																																																							

a larger initial memory consumption (twelve memory locations) because four processors are executing simultaneously, the average memory use is lower than that of sequential programming (seven locations compared to ten locations) due to dynamic memory allocation. In a large system this would mean that the storage not used in the latter stages of execution could be allocated to some other process, providing an overall improvement in memory utilization.

Figure 6.1.3.6(c) illustrates the memory-optimal resource allocation of the expression. Minimum memory consumption requires simplex processing, because each processor requires storage space for its results. Therefore, the time required in the evaluation of the expression is seven execution steps, the same as required by sequential programming. However, due to dynamic memory allocation, the average memory used by the memory-optimal resource allocation is six locations, compared to the sequential programming which requires eleven locations.

Figure 6.1.3.6(d) is an ISDS/HOS resource allocation illustrating a possible trade-off between time and memory constraints. In this case, only two processors are used, compared to the four required for time optimality. The average memory use and execution time can be seen to be between the bounds of the time-optimal and memory-optimal configurations. In a large system this would be indicative of relaxing execution-time constraints to save the expense of using more processing units and more storage space while still providing a significant improvement in execution time over the memory-optimal configuration.

Resource Allocation Summary

The results derived from the simple example examined in this section generalize to large systems with even more significant implications. The possible configurations of complex systems are far more numerous and could not be evaluated manually. ISDS/HOS techniques, via the Resource Allocation Tool, enable automatic determination of the optimal configuration for a specified implementation. The key to the automation of the process is that every cause or effect within a system appears explicitly in an ISDS/HOS system specification.

When a computation is defined by sequential programming, for example, the availability of each operand is assumed by the processor when it executes an instruction and this must be guaranteed by the programmer. The fact that instructions must be arranged in the proper order so that execution may proceed correctly is a principle reason that programming skill is needed and is a source of programming error. In an ISDS/HOS resource allocation, the readiness of an operation for execution depends solely on the availability of its input operands. This is a condition that can be determined dynamically or specified statically in many possible configurations in response to other constraints. As a result, much greater flexibility and better performance can be obtained without implementation errors.

6.1.4 Structuring Executive

Systems operating in real time are subject to unpredictable external events which cannot be analyzed during static development. Processes also must be dynamically scheduled in response to real-time events.

It is the responsibility of the Structuring Executive to perform these functions in real time. This requires that the tool be target system resident. In effect, the Structuring Executive subsumes the functions of an operating system as well as performing resource analysis and target system reconfiguration.

The need for dynamic reconfiguration of the system-control map in response to external events can be illustrated by a simple example. Consider a system in which a single communications channel is shared by multiple users as depicted in Figure 6.1.4.1(a). This could represent a hardware multiplex data bus, a single-frequency radio communication system, or even a group of people talking around a table. Because the system has only one communication channel, only a single user can broadcast at any given instant. There may be one or more listeners, depending on the nature of the system, so some mechanism of control must exist to route the transmission to the proper listeners. This situation is illustrated in Figure 6.1.4.1(b) where the user B is broadcasting to one or more of the other users. However, when B has completed its transmission, and another user is to broadcast, then the system must be reconfigured. The case where D is given the transmission rights is shown in Figure 6.1.4.1(c). To control the reconfiguration, a reconfiguration system would be established so that the

system of Figure 6.1.4.1(b) and the system of Figure 6.1.4.1(c) would be input to the reconfiguration system as data. Because of the simplicity of this example, it is possible to analyze all the possible system configurations manually and to insure statically all interface and data integrity. However, the same requirements for dynamic reconfiguration in complex systems would eliminate the possibility of manually analyzing all configurations statically. Thus, there is a requirement for some means to reconfigure the system-control map in response to unexpected external events and to insure that all data accesses and module interfaces are consistent with the axioms of ISDS/HOS.

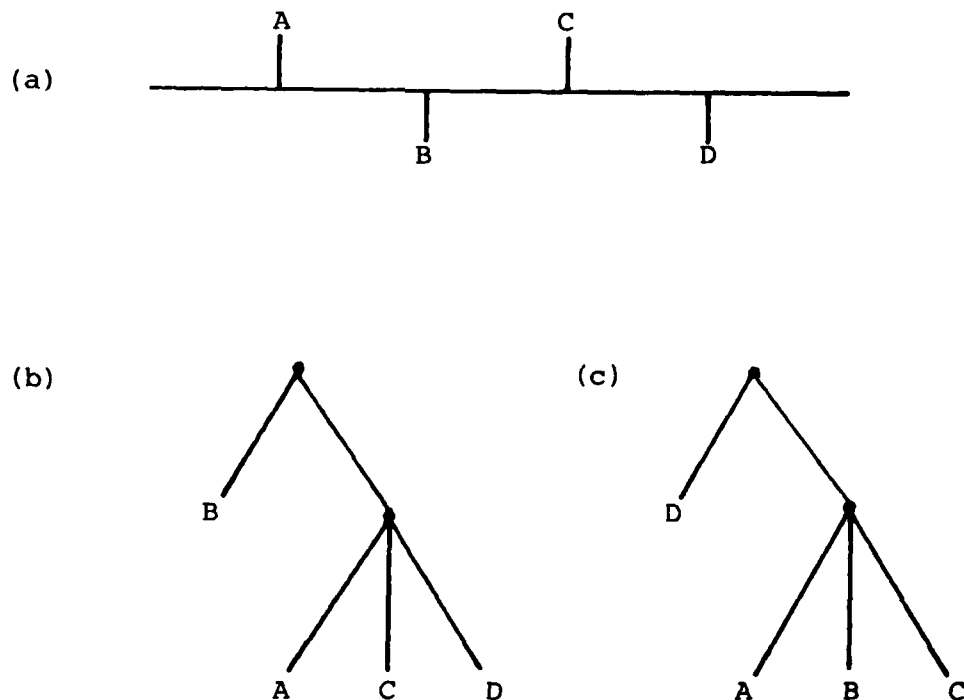


Figure 6.1.4.1
Dynamic System Reconfiguration

Functions of the Structuring Executive

The Structuring Executive will have the responsibility of providing real-time control of the target system operation. It must implement, in real time, multiprogramming and/or multiprocessing constructs. The functional requirement of the Structuring Executive is to handle aspects of system operation such as man/machine interface, hardware/software interface, error detection and recovery, real-time reconfiguration, dynamic resource allocation, analysis for timing and memory limitations, and an axiom analysis of the system in real time. Specifically, the Structuring Executive will: 1) control the ordering of those modules which can vary in real time dependent on operator selection; 2) assign priorities to processes based on the relative priority relationships, according to Axiom 6, for each control level; 3) prevent violations of the HOS axioms so that no two processes can conflict with each other; and 4) allocate the resources of the target system to maximize its utilization within safe operating limitations and prevent failure due to system overload.

The man/machine interface aspects of the Structuring Executive allow for a human to interact with a given system at any level. If sequences selected are not compatible, the Structuring Executive would detect an axiomatic error and automatically recover the system. The human can then select any reconfiguration of modules in real time without concern for introducing errors. For example, an avionics pilot would not need to memorize the order of a complicated crew selection list, since the software would provide automatic error detection and recovery.

The Structuring Executive consists of three components: the Dynamic Analyzer, the Dynamic Resource Allocation Tool, and the Dynamic Scheduler. These three processes interact in real time to realize the requirements for the Structuring Executive.

Dynamic Analyzer

The Dynamic Analyzer can provide a reconfiguration in real time by reordering of priorities based on the particular human or hardware inputs to the system. In the restructuring process, the Structuring Executive always maintains the relative timing relationships for all functions on a given level (and thus for the complete system) based on the fixed relative ordering defined for that level. Analysis can be conducted at the required module interfaces since the capability exists to provide timing, memory, domain and range limit requirements for selected modules in advance. For example, if the throughput of a process is larger than a given limit within a specific length of time, the Structuring Executive could terminate the process with the option to postpone its restart until favorable conditions exist or to selectively restart the process to include only its higher-priority functions.

Dynamic Resource Allocation Tool

Using a priori timing and memory limits specified for a model, the Dynamic RAT monitors the state of the system resources and allocates appropriate resources in real-time. This information is used by the Dynamic Scheduler to change the state of process queues. Essentially, the Dynamic RAT with respect to the static RAT is analogous to the interpreter with respect to the compiler.

Dynamic Scheduler

The Dynamic Scheduler enters a process into the process queues where its position in the queue depends on the current state of the system resources and the relative priorities of the tasks. The Scheduler within the Structuring Executive must implement the allocation decisions made by either a static or a dynamic Resource Allocation Tool.

Summary of the Structuring Executive

Conceptually, the Structuring Executive replaces the operating system with a resource-optimization tool for a multiprogrammed, multiprocessing environment which monitors, analyzes, and manages the target system in real time for maximum resource utilization and system reliability. To perform this function, the Structuring Executive must obviously be resident within the target system and as such, it will be an overhead process. While the purpose of the tool is to insure that the least possible waste of resources occurs, a trade off must be made, depending on the size and nature of the target system, between the relative services provided by the Structuring Executive and the overhead it requires. Due to the diminishing cost and weight of hardware and the relaxation of time-critical operation through use of multiple processing units, the overhead consumed by the Structuring Executive may become trivial compared to the operational enhancement and reliability it provides to a multiprocessing system. However, the prime concern of any system design is its performance, and particularly in simplex systems, processing-time consumption may be very performance critical. Thus, it will be possible to include in the Structuring Executive only those functions and only that degree of optimization power which the system designers determine to be performance justified.

6.2 Support Tools

The Support Tools available in ISDS/HOS will aid the disciplines of Management, Documentation, and Design. Some of the tools related to each of these areas are currently available or have been used in one form or another on various projects. The Management and Design tools when developed and used in accordance with the ISDS/HOS concepts provide more automated and useable tools.

6.2.1 Management Support Tools

Many of the current management reporting systems require massive amounts of information to be produced and entered into the data base manually. This effort hinders the utility of the system in that costs become prohibitive and the timeliness of the information impaired. It is, therefore, essential that the Management Support Tools be automatic where feasible in their data collection efforts and facilitate data entry where manual intervention is required.

Within the overall framework of the characteristics listed above, four Management Support Tools have been identified. These tools are (1) Data-Base Structure, (2) Resource Monitoring, (3) Collector, and (4) Inter-Revision Updater. These tools are conceived as automated aids to all levels of management throughout the system-development process.

Each of the tools listed above is conceptually described in subsequent sections.

6.2.1.1 Data-Base Structure Concepts

Inherent in effective management is timely access to pertinent information. In small development projects information can readily be obtained through informal methods. However, on large complex development activities, automated means of

storing, relating and retrieving information is required. The field of Data-Base Management Systems has provided numerous ways of storing, relating, and retrieving information efficiently.

The ISDS/HOS concepts utilize tree structures to illustrate the interactions of related components. However, any given node in the tree structure may be related to a number of different information sources (e.g. requirements, resources, software modules, hardware modules, statistics, etc.). The concept of network-oriented data structures is available to handle these multi-dimensional relations.

In ISDS/HOS three data bases will be of primary concern: requirements (as represented by an HOS functional decomposition), resource utilization, and component development.

Until a fairly comprehensive ISDS/HOS functional decomposition is available through the use of the ISDS/HOS specification language AXES, the ISDS/HOS Collector and Inter-Revision Updater are utilized to maintain and structure the requirements and their resulting specifications. Upon completion of the functional decomposition via AXES, the resulting control will be automatically used to structure the primary data base. All requirements and/or specifications for a given node in the decomposition will be placed in this data base.

The data base as generated from the decomposition will be related to the data bases for Resource Monitoring and Inter-Revision Updating as illustrated in Figure 6.2.1.1.1. The Data-Base Structure as illustrated presents an overview of the various data bases which comprise the core of the Project Management Support. There are several separate data bases which compose the foundation of the Project Management Support Tools. The relationships which can be drawn between these data bases give the flexibility and power required for effective management and efficient development.

In Figure 6.2.1.1.1, the Resource Utilization Data is structured by the Resource Monitoring tool which is described in this section. The Component Development Data Base is structured by the Inter-Revision Updater and the Collector, both described in this section.

Much of the data relations, illustrated in Figure 6.2.1.1.1, will be accomplished automatically. As a result, data collected in the various data bases can be correlated by the Resource Monitor to provide comprehensive project reporting. Where manual entry is required, interactive data entry will be utilized.

The structure and relations built into the Project System Data Base provide an integrated data base available to all levels of management in the development process. The ease of use of the system will facilitate timely reporting and access to the desired information.

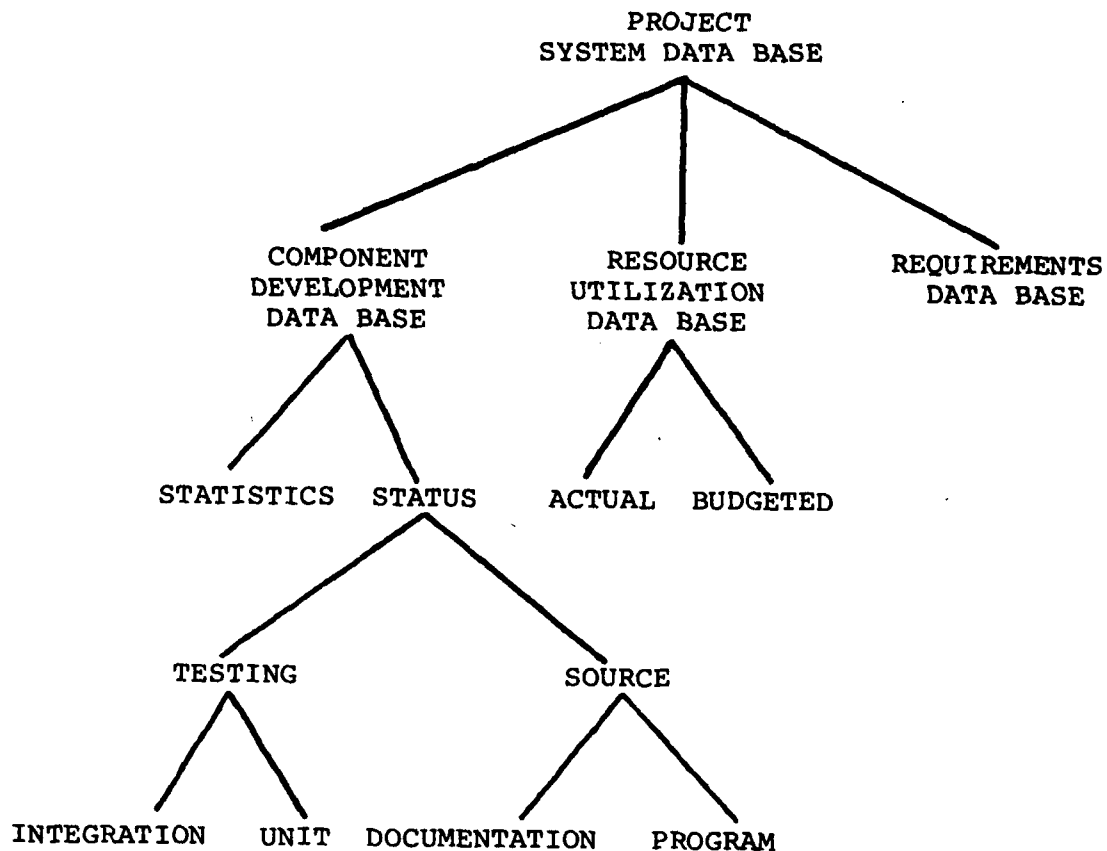


Figure 6.2.1.1.1
Structure of ISDS/HOS Project System Data Base

6.2.1.2 Resource Monitor

Effective management can only be accomplished through timely access to such information as projected and actual resource requirements. Among the more important types of information required are: (1) budget vs. costs incurred, (2) schedule dates and status, (3) manpower allocation and requirements, and (4) other resource utilization. Given timely and accurate data in the areas listed above, management can analyze problem areas* more realistically and take corrective action from an overall systems-development viewpoint.

Many data-management systems exist which can readily handle the information presented above. The processes of collecting and reporting such information, however, impair the utility of these systems.

Certain of the major problems associated with these processes are listed below:

- Data Availability:
 - Difficult to make estimates of resource requirements
- Data Entry:
 - Manual Process of data collection
 - Manual Process of data entry
 - Iterations of data entry required due to non-interactive edit checks

* In some cases, the information listed would be sufficient to detect problem areas. However, in general, problem areas are presented through other means, e.g. meetings, memoranda, etc. In these cases the information listed above could be used to determine possible alternatives for corrective action.

- Data Reporting:

- Untimely because of data-entry problems
- Voluminous reports
- Special reports require additional time with data becoming even more historic

The first problem area listed above, data availability, requires additional comment. In many of the studies conducted on software development (notably SAFEGUARD, MITRE, JHU/APL studies), the following problem was repeatedly stated: Managers find it extremely difficult to make estimates of manpower, budget, time and other resource requirements. Most estimates made of these requirements were based on "similar" development activities of the past. This problem area can be greatly facilitated through the use of ISDS/HOS and modern data-management concepts, as will be shown subsequently.

The other problem areas listed previously can also be alleviated through modern data-entry and data-management concepts, as will also be shown below.

To illustrate the above statements, a simplified ISDS/HOS functional decomposition map is shown in Figure 6.2.1.2.1. (The functional decomposition map can be created through the use of the ISDS/HOS specification language AXES.)

Resources (e.g. manpower, time, facilities, dollars, etc.) can be assigned to the nodes of the decomposition. For example, assume system S is estimated by top level management to cost five million dollars, and the break-down between sub-systems S_1 and S_2 is to be one and four million dollars, respectively. Further resource assignment occurs at each level during which lower-level and more direct technical experience is brought to bear on generating the estimates. As the funds are allocated down the tree structure, it may become

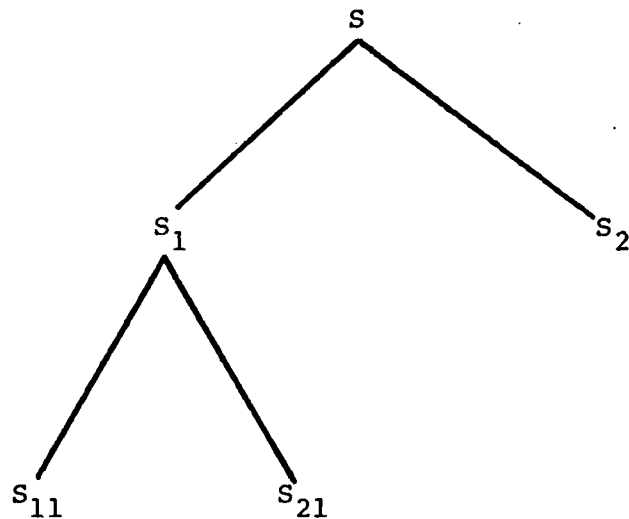


Figure 6.2.1.2.1
Simplified ISDS/HOS Functional Decomposition Map

apparent that a higher level of reallocation is required. In any case, once the funds have been allocated to the bottom level, one can automatically trace back up the tree to determine higher-level estimates. Iterations of this process will result in using sound technical and management skills in generating the final estimates.

Other resources can be similarly assigned and would probably be done simultaneously. Once all resources have been assigned, it would be possible to check their consistency at any level in the system.

To facilitate the above process, a data base would be created with the structure illustrated in Figure 6.2.1.2.1. Entering data would be performed interactively on a display terminal (e.g. CRT or storage tube) with a light pen or cursor. The user would simply place the light pen or cursor at the "node" of interest on the tree structure displayed on the screen. A data-entry program would query the user for the resource estimates and any supporting text. Inter-active edits would be performed automatically, and the data would be stored for the appropriate node. This method of graphical prompting and interactive data entry, edit and storage would greatly facilitate the data-collection process.

Data-reporting software would be similarly oriented: A graphical display would prompt the user to the specific area of the tree structure (system decomposition) of interest. Resource estimates could be obtained for a single node and could be traced up or down the tree. Output could be graphical, in the form of Figure 6.2.1.2.1 with the resource estimates indicated at the nodal (function) points, or tabular.

During the project-development process, one would enter incurred resource utilization (funds, manpower, computer time, schedule status, etc.) identical to the method described above.

Inter-active edits would be performed and the data entered into the data base automatically.

The output would be available interactively or as an inter-active request for graphical or tabular hardcopy. The format of Figure 6.2.1.2.1 could be used to indicate reporting period or project-to-data resource utilization. Time-line graphical presentation of budgeted and actual resource utilization could be presented by individual resource by node (function) at any level.

Changes to any resource allocation (e.g. funds, schedule, manpower, etc.) would be entered as indicated above with any supporting text and would be automatically retrieved and reported for any related queries.

The Resource Monitor would be used to correlate statistics collected by the Inter-Revision Updater with resource data to provide detailed management reporting.

The emphasis for the Resource Monitor has been to facilitate management access to resource estimates or utilization in a very timely manner and to any level of detail required. The Resource Monitoring Tool is built on an ISDS/HOS structure utilizing state-of-the-art data-entry, data-management and data-presentation concepts.

6.2.1.3 Inter-Revision Updater

As component software modules are completed to the satisfaction of the software-development personnel, they are turned over* to the appropriate Assembly Control Supervisor along with test-data input and execution command language. At this time, these component software modules become "frozen". These modules

* The act of "turning over" could be simply placing a read-only lock on the source-code, data-input, and execution-command language files.

are then ready for unit testing, integration, and system-interface testing.

Any subsequent changes to the software modules will be made by accessing the prior version of the source-code (or test data-input or execution-command language) file and entering changes. The file (for source code, test-data input or execution command language) for a given version will be maintained separate from prior versions, such that for a given version only those changes to the immediately preceeding version are saved. When a version becomes frozen, the file containing the changes to the preceeding version becomes frozen. Figure 6.2.1.3.1 illustrates the original source code with sample commands creating two subsequent revision files.

The Inter-Revision Updater will automatically supply revision numbers and append them to the module names. The Inter-Revision Updater will maintain the original source-code, test-data input and execution-command language files and all revision files. A reference to a software module will automatically assume the latest changes unless an earlier version was specifically requested. The use of an Inter-Revision Updater facilitates concurrent use of software modules by separate organization (e.g. testing, integration, development), and assuring each that the current version is used.

Table 6.2.1.3.1 lists the files maintained by the Inter-Revision Updater. All these files would be handled as described above.

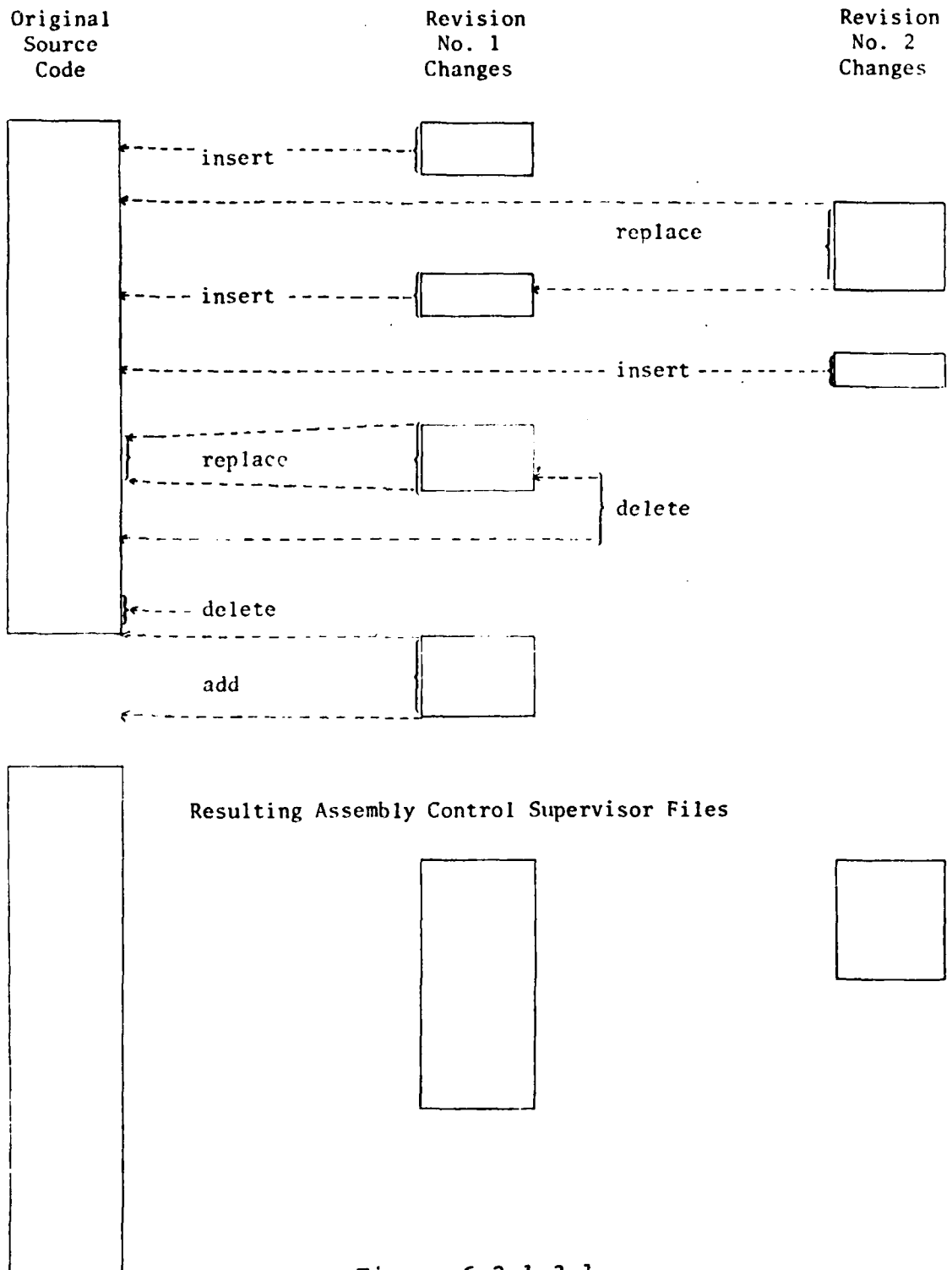


Figure 6.2.1.3.1
Inter-Revision Updater Example

Table 6.2.1.3.1
Files Maintained by Inter-Revision Updater

source code test-data input execution command language documentation object code	The original file and <u>each</u> revision will be maintained as separate entities.
statistics	This file will be updated throughout development and testing

Once a module has been frozen, the ACS would compile or assemble the module and store the resulting object module in an object-module library. This library along with the execution command language and test-data input for a given version will be used by the ISDS/HOS Collector.

The statistics files would be automatically created and updated by the Inter-Revision Updater. The statistics which would be collected automatically would include those listed in Table 6.2.1.3.2. These statistics could be queried by managers to determine status of the various component software modules, and to get a better "feel" for the utilization of resources.

The Inter-Revision Updater will provide the development personnel and the Assembly Control Supervisor with a coordinated software development controller in the interim ISDS/HOS environment. In this capacity the Inter-Revision Updater will automatically indicate which of the files (see Table 6.2.1.3.1) pertaining to a given software module have been revised. The

ACS can then verify that the unchanged files are compatible with the changed files (e.g. keep the documentation or test-data input files up-to-date with the source-code file). In addition, the Inter-Revision Updater can keep track of which tools (e.g. Structured Design Diagrammer) have been used in conjunction with the development of a software module. Software-module documentation could be produced using the files maintained by the Inter-Revision Updater.

Table 6.2.1.3.2
Statistics Automatically Collected by the
Inter-Revision Updater

Number accesses
Number lines source code
Number lines source code by revision
Compilation attempts
Error-diagnostic summary
Interactive connect time used
Computer time used

6.2.1.4 Collector

When a large number of software modules are developed incrementally as components for a system, the job of assimilating these modules becomes prohibitive and error-prone. In ISDS/HOS this job would be done automatically by the Collector.

The Collector would operate from a control map. For purposes of illustration, refer to Figure 6.2.1.2.1. Assume that system S_1 is to be executed. The user would enter a command to the Collector to collect and execute system S_1 . The Collector would trace down the tree from S_1 to the bottom-level nodes collecting as it goes the object-module (compiled or assembled versions of the source code), the execution-command language, and any required test-data input files. (The Collector would utilize the Inter-Revision Updater in performing this task.)

If the user wanted to modify any test-data input in the test-data input files, simple commands would be issued to override the desired data items in the appropriate file. (The Collector would utilize the Inter-Revision Updater to perform this task.) These overrides would be temporarily stored for the given run and documented automatically on any resulting output.

Similarly, if the user desired versions of modules other than the latest, the appropriate revision number for the specified module would be entered. The Collector would retrieve that version and its associated execution command statements and test-data input. Prior to establishing the run, the Collector would verify that the interface would match for the specified version. Error diagnostics would be interactively produced and listed on any resulting output.

Furthermore, if the user desired to create a new system, the Collector could be used to automatically assemble the components of the desired system. For example, assume subsystem S_1 is a simulation with system S_{11} being the environment and system s_{21} being the target system. The user has constructed a subsystem S_3 which is an error-analysis system. The user could automatically construct the new system S_0 shown in Figure 6.2.1.4.1 by entering commands which: 1) establish the node S_0 ; and 2) link nodes S_1 and S_3 with S_0 .

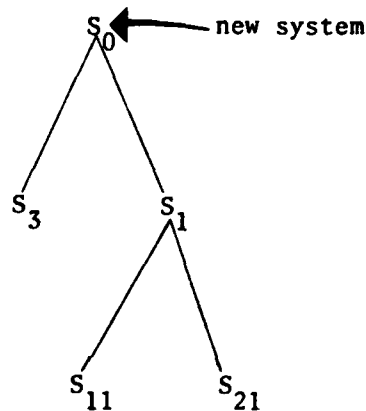


Figure 6.2.1.4.1

User Creation of New System via the ISDS/HOS Collector

The Collector would check subsystem interfaces and indicate any inconsistencies. This is accomplished in the following manner: All subsystems (or software modules) would contain as a special instruction a description of its interfaces with external modules (e.g. number of data items and data type for

each parameter).^{*} The Inter-Revision Updater would store these special instructions as part of the Execution Command Language files when the module became frozen. The Collector would check these interface instructions automatically when assembling a system or creating a new system.

The Collector would assemble any error diagnostics and computer statistics and update the statistics file maintained by the Inter-Revision Updater for the system being collected.

The ISDS/HOS Collector is a tool available to an Assembly Control Supervisor or Testing Team for assembling a system configuration interactively. Interface analysis would be accomplished interactively thus eliminating lost time, effort and funds for assembling incompatible subsystems.

The statistics collected by the Collector provide an immediate view of configurations tested and the success or failure of the tests conducted. The Collector used in conjunction with the Inter-Revision Updater will save valuable time in the development, integration, and testing disciplines of the systems-development process.

6.2.2 Documentation Support Tools

The data bases or files that pass from phase to phase in the development process will contain information in various formats. Files will contain requirements, specifications, personnel and resource information, documentation, and for the immediate

* The Compiler could be used to automatically create these instructions when a reference to an external software module occurred.

future, source code. The documentation support tools described in this section will enable the users of ISDS/HOS to build and maintain these data bases conveniently and efficiently. The current use of such tools (text editors and formatters) are for program and documentation development. In ISDS/HOS they will be used to input the files of requirements, AXES Specifications, and resource information. These tools will also be the vehicle for interacting with the Management Support tools described in the previous section. The use of these interactive documentation support tools is straightforward; the user accesses the host system via an interactive terminal to build a file, to verify its contents, and then stores the file in the file management system. He can later call for that file and update or revise it to reflect new information. When desired, he calls the file from the file management system and requests printed output in a selected format.

6.2.2.1 Text Editors

A powerful interactive text editor is an essential tool for program, data, and documentation development. In addition to providing powerful techniques for program, data, and documentation preparation, the editor must have a relatively simple syntax that is easy to learn and must provide the novice with some prompting to help him avoid classic beginner errors.

Desirable editing capabilities:

- addressing - The editor should provide the ability to address by absolute line number, relative line number, and by context. The editor should allow compound addressing (combining any of the other three techniques) and addressing of a series of lines.
- inputting - The editor should provide the ability to enter text from the terminal or from the file system into any position of the buffer.

- editing - In addition to the ability to print, delete, locate, or change a line, the editor should provide the ability for a global substitution of one character string for another.
- interacting - The editor should allow the user to pass commands to the command environment external to the editor.

This list of desirable characteristics is not necessarily complete; nor is it meant to eliminate existing editors from consideration.

6.2.2.2 Text Formatter

The text formatter will allow the user to type out text files in manuscript form. This facility greatly enhances the user's documentation ability. A user will prepare a file of text lines and control lines as input to the formatter to be output on a device and in a format as he prescribes. By using the editor and formatter, the user can extract documentation from his program for inclusion in a user's guide to be printed in manuscript form.

Formatters of this type are often used by the clerical staff of a project as well as by the engineers. The formatter must be a powerful tool, yet be simple to use and easy to learn.

6.2.3 Design Support Tools

The Design Support Tools to be available in the ISDS/HOS environment are simulators, performance monitors, and emulators. As described in subsequent sections, each of these tools is designed and developed in concert with the ISDS/HOS concepts. These tools provide a comprehensive set of tools to be used in studying systems under development.

6.2.3.1 Simulator

"A simulation of a system...is the operation of a model or simulator which is a representation of the system...The model is amenable to manipulations which would be impossible, too expensive or impractical to perform on the entity it portrays. The operation of the model can be studied and, from it, properties concerning the behavior of the actual system or its subsystem can be inferred." (SHU60)

In a systems-development process, simulators provide engineers and designers with a tool to study the system under development and its interactions with external influences. Within the context of ISDS/HOS, simulators are computable systems; they can, therefore, be designed and developed in accordance with the axioms of ISDS/HOS.

Figure 6.2.3.1.1 illustrates a top-level ISDS/HOS decomposition of a simulation into two major functions: the system under study and its simulator.

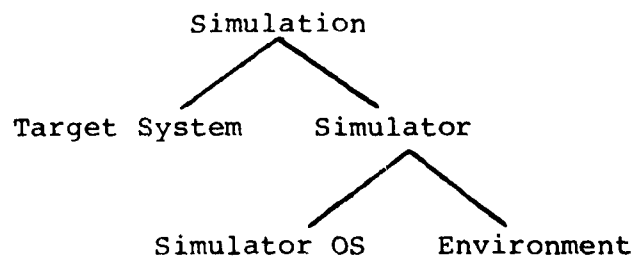


Figure 6.2.3.1.1
Top-Level ISDS/HOS Decomposition of a Simulator

Simulators are generally used throughout a systems development process: from a feasibility demonstration of the system to be developed early in the process to a full scale simulator which is carried throughout the development process and is continually refined with more detailed models. The ISDS/HOS concepts of software development lend themselves readily to the process of simulator development. For each new project, specific environment modules are developed in order to interface in simulations with a specific target system.

Several characteristics of an ISDS/HOS simulator are required. These characteristics, listed in Table 6.2.3.1.1, provide users of the simulator added capability to study the system being developed, the environment being simulated, and the interaction of the system with the environment.

6.2.3.2 Performance Monitors

During design, development and operation, systems are monitored for resource utilization and anomalous behavior by Performance Monitors. The function of Performance Monitors is to compare a priori performance statistics with performance statistics collected in real-time (via simulation or operation) to detect anomalous behavior. Examples of these statistics would be:

- excessive calls to specified routines or executions of specified instructions
- number of interrupts outside of the normal bounds for a given time period.
- accessing protected areas of memory
- inconsistent clock increments

TABLE 6.2.3.1.1
CHARACTERISTICS OF AN ISDS/HOS SIMULATOR

TIME-ORIENTED FEATURES:

- Clock - simulated clock with varying degrees of fidelity.
- Maxtime - maximum time which simulation is to run, after which simulation will be stopped.
- Modification of module execution times and frequency of module execution during simulation run.
- Modification of simulation fidelity (clock frequency).
- Standby simulation, i.e. placing modules in a wait state for specified time or until an event occurs.
- Time Advance - Advance simulated time to a specified time or by a specified time increment, or until the next event and update all parameters to that time.
- Time compression to determine impact of execution-time loss in simulated system.

PROGRAM- AND DATA-ORIENTED FEATURES:

- Data initialization - simulation parameters can be initialized to specified or randomly generated values for simulation checkout and simulation reproduction.
- Data modification - variables and parameters can be modified during a simulation run at a specified time, event, or execution location.
- Program patch - temporary inclusion at a specified location of a program patch to facilitate checkout.
- Reserving and releasing data areas (protect enabling and disabling).

DEBUG-ORIENTED FEATURES:

- Check status of simulation parameters (refer to Data Initialization & Data Modification above); parameters, their values and their status would be listed.
- Trace of execution flow beginning and ending at specified locations, times or events.
- Printout of simulation parameters as specified.
- Rollout - simulation status and all parameter values would be stored on secondary media (e.g. disk or tape) for subsequent startup as separate simulation run. Rollout could occur at specified times, events or locations.

TABLE 6.2.3.1.1
(continued)

SIMULATION-CONTROL FEATURES:

- Start - start simulation at specified time, event or location. Start condition may involve initialization of simulation using data stored as a result of the rollout feature described above.
- Stop - stop simulation at specified time, event or location.
- Message - print out pre-specified message at specified time, event or location.
- Insert simulation commands into simulation input stream from pre-stored file.
- Deactivate simulation commands at specified time, event or location.

OUTPUT-CONTROL FEATURES:

- Print out all or specified simulation parameters and control values in a formatted listing.
- Print out summary simulation parameters.
- Print out all or selected simulation parameter and control values at specified times, time intervals, execution locations or events.
- Print clock times at simulation increments.
- Plot specified simulation parameters.

PERFORMANCE MONITOR-INTERFACE FEATURES:

- Enable an abort or a refresh/restart of the simulation on the occurrence of certain events, such as:
 - excessive repetitive execution of an instruction or an instruction sequence.
 - access into protected memory areas.
- Enabling memory dumps and parameter dumps upon abort or refresh/restart.
- Resource utilization checks.

During the design and development phases, the a priori statistics would be estimates provided by engineers and designers. These statistics would be compared with similar statistics collected by the Performance Monitor during simulation or emulation execution. If an anomalous condition was observed, the Performance Monitor would stop the simulation or emulation and produce diagnostic information (such as warning or error messages) and debug aids (such as memory dumps or parameter listings). Depending on the severity of the error, the Performance Monitor could restart or terminate the simulation or emulation.

The use of a Performance Monitor in the design and development phases provides a more accurate method of developing resource-utilization statistics. This would be accomplished via hardware attached to the host (or target) machine or software monitors, which would increment counters for various instruction classes as the related instructions were executed. The use of these counters would give a fairly realistic estimate of processing time required. Similarly I/O requests, I/O transfer times and idle time could be obtained.

Refined statistics derived from simulations or emulations would be used as guidelines for the real-time Performance Monitor which will be a sub-function of the Dynamic Resource Allocation Tool in the Structuring Executive. The operation of the real-time Performance Monitor would be the same as described above, except in the case of anomalous conditions. If an anomalous condition arose, the Performance Monitor would signal the Structuring Executive to initiate system recovery procedures.

6.2.3.3 Emulators

"An emulator may be defined as hardware, microprograms, and software added to one computer system to enable it to execute programs written for another system." (MAL75).

Once the requirements for a computer system have been defined, they can be represented in the ISDS/HOS specification language AXES. AXES would then produce a functional decomposition of these requirements, in which the bottom level of nodal families represents primitive operations (specifications for the machine instruction set) and primitive data types (data types of the machine). Using this process, one can define the specifications of a target machine, including the required instruction set and data types.

In order to study the target machine and do performance analysis, it is necessary to simulate the target machine. In some cases, the instruction set of the host machine may not be adequate to perform the instruction set of the target machine. In almost all cases simulation of the target machine would be an inefficient process. As a result, an emulator of the target machine would be developed for the host machine.

An emulator consists of a mixture of software and firmware which would "simulate" (emulate) the target machine on the host machine.

In developing an emulator, the designers and engineers must specify the mapping of the target-machine instructions onto the host-machine instructions. This mapping constitutes the requirements for the microcode. In ISDS/HOS these requirements would be formulated in AXES. This formulation would result in a functional decomposition, the bottom level of which would be the primitive operations (machine instructions) and the data types of the host machine. This decomposition would

be analyzed (by the Analyzer) for interface consistency and then passed through the Static Resource Allocation Tool to obtain an optimal configuration of the decomposition tree. The result of this process would be the optimal specification (target-to-host mapping) for the emulator. Figure 6.2.3.3.1 illustrates the procedure described above.

The Static Resource Allocation Tool would output the sequence of host-machine primitive operations (instructions) required for a target-machine primitive operation (instruction).*

The required execution time (on the host machine) for each target-machine instruction as represented by a sequence of host-machine instructions could be obtained. By using an assumed instruction mix (i.e. frequency of occurrence for each target-machine instruction) and the execution time for a host-simulated target-machine instruction, one could determine which target-machine instructions would be more efficient to implement in microcode. The remaining target-machine instructions could be simulated on the host machine.

The result of the process described in this section would be the development of an optimally efficient emulator with reliability inhibited only by the manual process of mapping target-machine operations onto host-machine operations.

Once developed, the emulator would operate similarly to an interpreter. For further insight into this aspect, refer to Section 6.3.7.

* In some cases, it may not be possible to map a target-machine instruction onto host-machine instructions, e.g. in the case of certain input-output device instructions. The output of the Static Resource Allocation Tool would be the specification for the hardware/firmware to emulate these instructions.

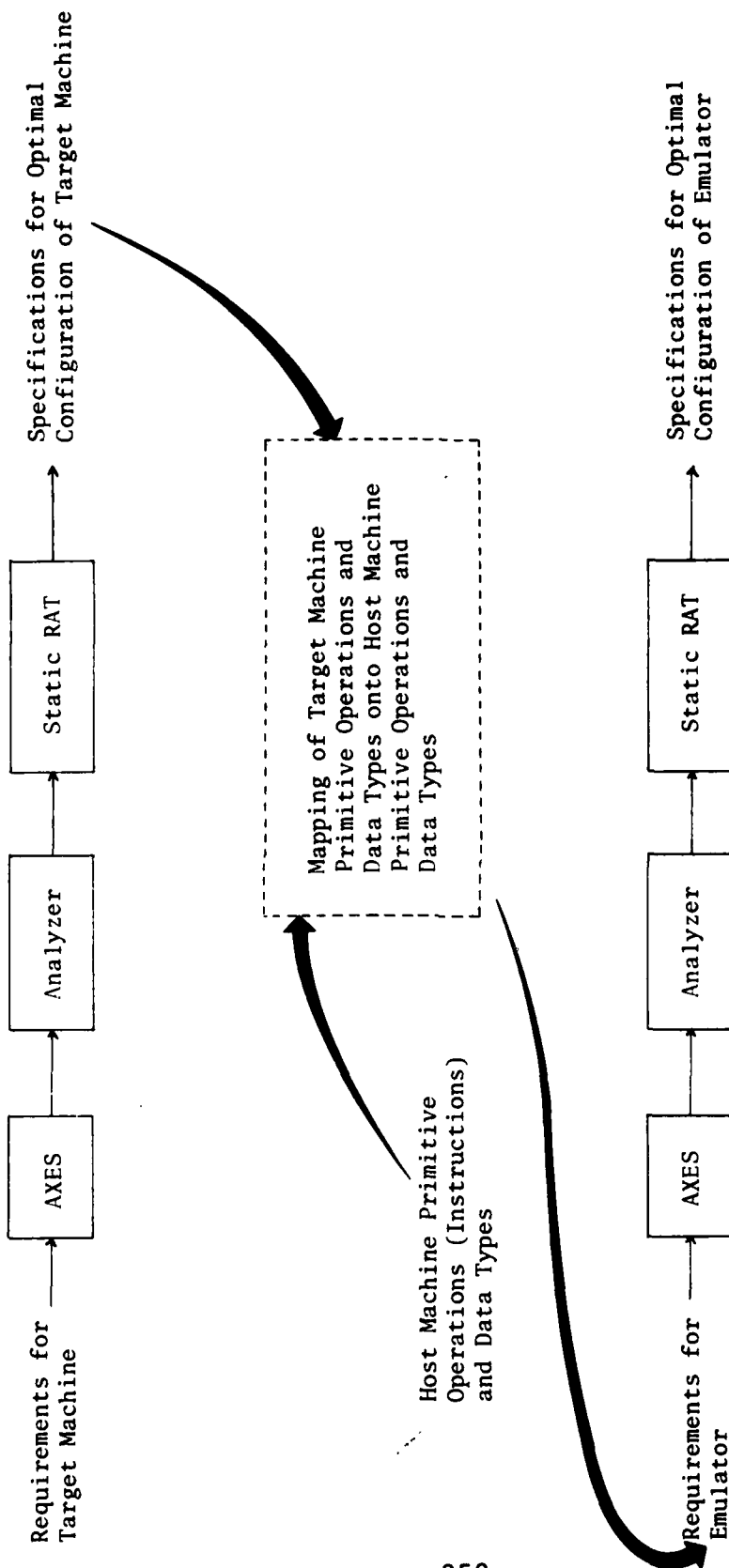


Figure 6.2.3.3.1

Summary of Procedure for Producing Optimal Specifications for Emulator

6.3 Incremental Tools for Current Use of ISDS/HOS

The tools presented in this section provide for an incremental approach toward adhering to the ISDS/HOS concepts. These tools are generally available; however, conceptual descriptions are provided which place requirements on these tools in order to make them consistent with ISDS/HOS concepts. Therefore, modifications of the available versions may have to be performed in order to enhance their reliable operation. Reliability in the use of these tools cannot be guaranteed due to their non-automated use; however, an approach toward gaining reliability can be attained by following the guidelines presented for the use of the various tools. The tools, as described, will provide an evolutionary trend toward the objective of ISDS/HOS.

6.3.1 Assembly Language

Although the ISDS/HOS interim environment centers around the use of an HOL for program development, it is conceivable that a particular customer could require the programming to be done in an assembly-type language. Historically, requirements to program in an assembly language have been made to improve efficiency in terms of space and time of the executing code. This efficiency argument is far less impressive when one considers the rapid decrease of hardware cost and size, in relation to the continuing increase of programmer cost. The reliability argument is that it is much simpler to verify HOL code than assembly language. ISDS/HOS discourages the use of assembly languages; they are covered here because the possibility of their use has not been eliminated.

Since assembly languages generally correspond one-to-one with the machine code and have full access to the instruction repertoire of the machine, it is difficult to describe charac-

teristics of an assembly language. Such a description would, if pertaining to the instruction capability or performance, apply also to the machine.

Static verification appears to be infeasible considering the large variety of "tricks" assembly-language programmers have employed. There are no known techniques to trace variable usage statically, to determine the destination of transfers statically, or to restrict the usage of variables across interfaces for straight-line assembly code. Therefore, the decision to use an assembly language should be made in light of the explicit trade-offs between reliability and code efficiency.

To be used in the interim ISDS/HOS environment, assembly language must have the ability to create and reference macros. The basic idea of macros is text substitution or insertion: an identifier in the source program is replaced by a string of characters from some other character string. These macros can be created in advance and stored by name. Other assembly language programs can then include these macros by simple reference to the name. These macros can be used to insure interface consistency within the resource-allocation process made by the user.

Admittedly, this takes away from the assembly programmer's tricks, and, therefore, the program manager should consider this fact if he requires that assembly language be used.

The use of macros for module interfaces is the only way to adhere even remotely to the concepts of ISDS/HOS when an assembly language is used for implementation.

6.3.2 Macro Processors/Assemblers

"An assembler is a program which translates a source program written in assembly language into the machine language of a computer." (GR171)

A macro processor may be implemented to either substitute assembly-language code or machine code for a specified character string. An assembler, as used in the interim ISDS/HOS environment, would differ from a traditional assembler by including features to improve the reliability of code. Reliability enhancements to existing assembler techniques include:

- checks that control transfers are not made into data
- automatic symbolic labeling for control points ("branch to" points), if not provided by programmer, with appropriate flag on output listing

The macro processor is most commonly used as an extension to the basic assembler. A macro in its simplest form is a one-line abbreviation for a group of instructions. The macro processor scans over the text searching for macro definitions and macro references, and then attempts to resolve these statements. If statements cannot be resolved (e.g. assembly-language statements), the input string is added to the output string without change; if the statement can be resolved, the input string is replaced by the specified output string.

In order to make macro processing more flexible and to relieve the programmer of redundant coding, macros may be referenced with or without arguments. As an example, a given computer environment may have a standard set of instructions to perform a certain task, such as pass data between subroutines. In order to avoid coding the same instructions repeatedly, the CALL macro would map into the sequence of assembly-language statements needed to pass arguments between subroutines.

The macro processor, as used in the interim ISDS/HOS environment, will differ from existing macro processors in the macro-definition phase. Macros will be built on macros by program-design engineers and be made available to programmers. Primitive macros such as "+" and "=" will be used in more sophisticated macros. Programmers will not be permitted to define their own macros. Design engineers will define macros in accordance with ISDS/HOS axioms.

Programmers who use macros in place of in-line code will benefit, first, by ease of programming and, second, by confidence in the reliability of the code supplied by the macro processor.

6.3.3 Higher Order Language (HOL)

The implementation language, as used in the interim ISDS/HOS environment, will be based on the HOL recommended by the DoD High Order Language Working Group (HOLWG) chartered in January 1975. Without attempting to critique thoroughly the "TINMAN" version of "Requirements for High Order Computer Programmer Languages", March '76, we can say that the eventual HOL must be examined for compliance with ISDS/HOS concepts and axioms. Certain structures of the language may have to be

eliminated and others added, but the present state of the working group's efforts indicate that the final product will be acceptable for the Implementation Language. Some of the particular language features to be considered in examining an HOL for ISDS/HOS use are covered here.

A HOL program shall consist of a set of procedures. Each procedure shall have an input list and an output list that specifies all of the arguments of the procedure. A procedure shall have no access to an object unless the object appears in the input or output argument list of the procedure or unless the object is a local variable of the procedure.

A very important consequence of this clear separation of inputs from outputs is that an HOL procedure cannot produce side-effects. This means that it is not possible to write an HOL procedure whose execution can produce a state change in any object that does not appear in the procedure's output list.

A HOL procedure cannot produce partial outputs. This means an invocation of a procedure produces no state change in any of its outputs until it produces all of its outputs. The following example illustrates a way in which this might be accomplished

```
(a,b,c) = f(x+y, x);  
procedure f(p,q) returns (integer, real, real);  
.  
.  
.  
return (25,7.2,p-q);  
end f;
```

The top line of the example is a call on the procedure f which returns three values.

The inability of a procedure to produce partial outputs allows recovery from procedure failure. Failure of any component of a statement will cause failure of its containing procedure. Failure of a procedure means that none of its outputs have been produced. Consequently, failure of any operation within a procedure leaves the caller of that procedure in the same state that it was in prior to the call. The definition of a procedure call will allow the caller to detect and respond to failure of the called procedure, without any of the elaborate facilities of PL/1 or more complicated facilities (G0075).

A very important consequence of the separation of inputs from outputs and the inability for a procedure to produce partial outputs is that input arguments do not have to be copied. An input array argument can be passed by address without fear that some element of the array will be changed by the procedure to which it is passed during evaluation of that procedure.

The HOL shall contain facilities for the creation and control of real-time processes. These facilities shall be sufficient for control of real-time processes using one or more processors. As in the ILIAD language, deadlocks will be impossible because all global objects will be associated with a boolean valued semiphor called a lock, and because a process will be unable to access a global object unless it has first locked the object. Furthermore, all global objects that are accessed by an operation must be locked prior to the start

of the operation.

In addition to the basic locking constraint, the HOL shall contain facilities for: starting a new process, stopping a process, delaying a process, changing the relative priority of a process, and testing the locked/unlocked status of a set of global objects.

The control structures of an HOL shall be limited to the D' control structures of Marcotty and Ledgard (LED75) (While, Until, For, Case).

Input/output facilities of the HOL shall be suitable for use with a wide variety of devices and/or file structures, including devices found in typical tactical applications.

An HOL procedure can be declared to be an overlay procedure. When an overlay procedure is called, it will be loaded into main storage, where it will remain until it is displaced by another overlay procedure. This facility provides an efficient and easy-to-use form of demand paging of procedures. Overlay procedures will be defined in an implementation-independent manner that ensures that program logic will not be affected by the transfer of a program to another host system.

The HOL shall facilitate separate or combined compilation of multiple procedures.

The HOL will facilitate the compilation of optimized object code due in large part to the separation of inputs from outputs and due to the elimination of side effects.

Type checking will be total and will be performed at compile time as well as during the linking of separately compiled procedures. User-defined types will be allowed in a manner similar to that of ALGOL-68 and PASCAL.

Type checking of user-defined types will not require combined compilation of all procedures that use a given type. However, the compiler will use a type-definition library to ensure that a given type is always encoded in the same manner.

An HOL object procedure will always be re-entrant. On the ISDS/HOS host, they may also be shared so that multiple users will be able to execute the same object procedure simultaneously.

It is anticipated that the implementation language will differ to some degree from the HOL recommended by the HOLWG. To accommodate those differences, the programs developed in the ISDS/HOS interim environment will require some amount of preprocessing to be acceptable to the HOL compiler.

6.3.4 Compilers

It is likely that existing compilers will be used in the initial ISDS/HOS environment. The HOL section discussed features that will require preprocessing before compilation when they are included in the HOL. A further step toward reliability is to develop a compiler for the prime HOL of the interim ISDS/HOS environment. By developing this compiler using incremental ISDS/HOS techniques, the translation process will be more reliable than an existing compiler with preprocessing. The "TINMAN" version of "DoD Requirements for High Order Computer Programming Languages" discusses some desirable features of compilers specifically in relation to reliability and portability.

Reliability

Reliability is of primary concern for compilers in the interim ISDS/HOS environment, both in the production of object code and in detection of errors. Many current HOL compilers are known to produce object-code errors; that is, certain combinations of HOL statements when compiled into object code do not carry out the intended logic.

The "TINMAN" documentation, referenced above, specifies that error diagnostics and error conditions should be covered in a description of the language. This is a desirable feature, and any additions to the HOL should be accompanied by appropriate error information. In addition to the language-specified syntactic and semantic errors, the compiler should examine the interfaces for consistency. Although the HOL will not permit blanket enforcement of the axioms, there are some interfaces, specifically procedure calls, that can be checked for access rights of variables. Interface errors will be detected and classified according to severity, and an error diagnostic will be emitted in the output listing.

Portability

The compiler design will separate the machine-dependent functions of code generation from the machine-independent functions. This will allow compiler portability to other computers by isolating the compiler modules dependent on that computer.

6.3.5 Structured Design Diagrammer

Documentation has historically been regarded as a chore by programmers and, as a result, is frequently de-emphasized in the interest of producing code. The documentation produced is often out of date, erroneous, of poor quality, or organized in widely varying formats. Accurate documentation is often produced only as a final step in the programming process and is thus unavailable throughout the development cycle. This lack of, or delay in, documentation leads to communication problems in coordinating the efforts of individual programmers within a group.

The Structured Design Diagrammer, as an integral tool of interim ISDS/HOS, will provide automatic documentation of computer programs illustrating program flow, data intersections, and embedded comments.

Automatic documentation provides a mechanism for maintaining positive control over rapidly evolving programs. Documentation is produced immediately by the Structured Design Diagrammer for every revision of each program module. The documentation is uniform in format and accurately reflects the organization of the code. Best of all, the production of the documentation requires minimal effort on the part of the programmer.

In addition to automating an historically manual and tedious task, the Structured Design Diagrammer outputs a flow-chart superior in format to conventional flowcharts. For example, Figure 6.3.5.1 (MUL72) is a conventional flow diagram. Figure 6.3.5.2 (HAM73) is basically the same design but with structured notation replacing conventional flowchart symbols. A lot of time and effort is necessary to understand all the algorithm paths in Figure 6.3.5.1. An interruption

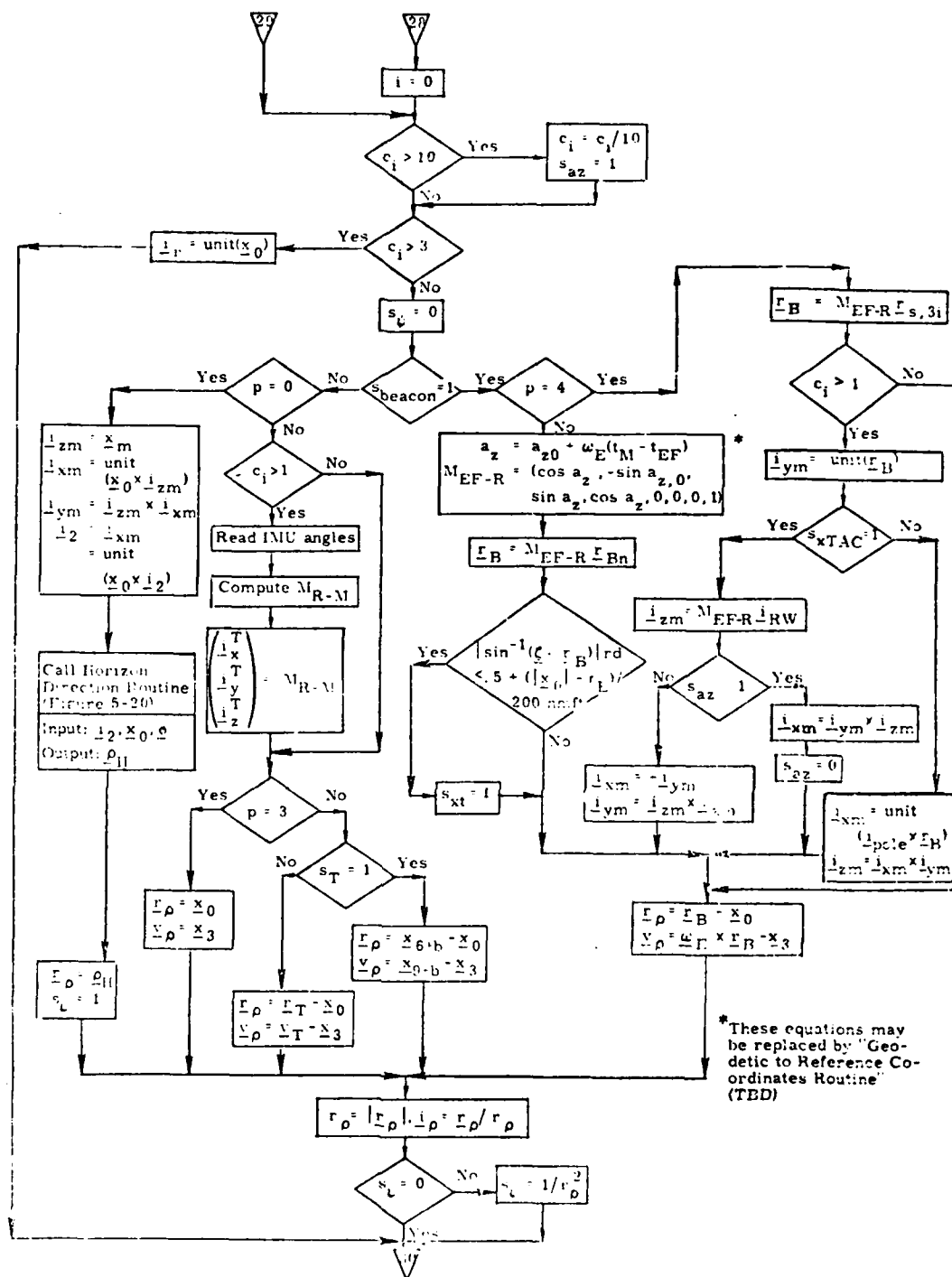


Figure 6.3.5.1
A Structured Program Using Standard Symbols
To Show Flow of Program Execution

forces the reader to start from the beginning. Consider Figure 6.3.5.2. Notice that after each logical step has been completed, control returns to the main flow of the algorithm. The nested decision levels are clearly distinguished, and it is easier to see the state of the system at each node.

The Structured Design Diagrammer functions as a design aid to the programmer and a control aid to the manager. The linear sequence of program instructions provides assistance to the programmer in obtaining:

- a) an optimal path to minimize the number of logic test cases, and
- b) equivalence comparisons between the proposed design of algorithms.

The preparation of structured design diagrams has immediate advantages to the user of a programming language: nested decision levels can be clearly represented; main algorithm flow is more visible; assumptions with respect to data can be separated from the actual data provisions.

The Structured Design Diagrammer requires the source-program input to be organized in the structured form. Violations of this structure are detected and flagged. This tool may be used both to train programmers in the application of structured-programming concepts and to enforce the use of these concepts.

The uniform standards and conventions of the diagrams reduce the effect of major differences in design approach of different engineers, groups or organizations working on a common program. From a management point of view, the automatic documentation produced by the Structured Design Diagrammer represents a valuable mechanism for ensuring communication between groups and for coordinating the programming effort.

Structured Notation

The basic unit of a structured design diagram is the "block". The "block" is a module which has a single entrance and a single exit.

The algorithm flow of the program depends on the decision structure, where the object of that decision can be thought of as a submodule to the decision statement. The statements used to make decisions are the basic structured statements shown in Figure 6.3.5.3.

Inherent in this representation is the knowledge that any decision statement performs the required submodule resulting in the main program flow. This is not available with conventional flowcharting techniques.

In addition, the structured design diagram notation presents a more adequate representation of a CALL in that it recognizes that the main purpose of any CALL is to manipulate data flow. Associated with each CALL, therefore, is the data module which presents the intersection of the data used by the calling program and the called program (Figure 6.3.5.3).

The definition of the data module assumes that the overall program structure has been completed and defined elsewhere. For structured design diagram notation, only the location (e.g., COMPOOL, local, etc.) and organization (e.g., matrix, array, etc.) must be specified for each data element of the data module. The program documentation in this form illustrates the data flow and control flow to the reader.

STRUCTURED DESIGN DIAGRAMS

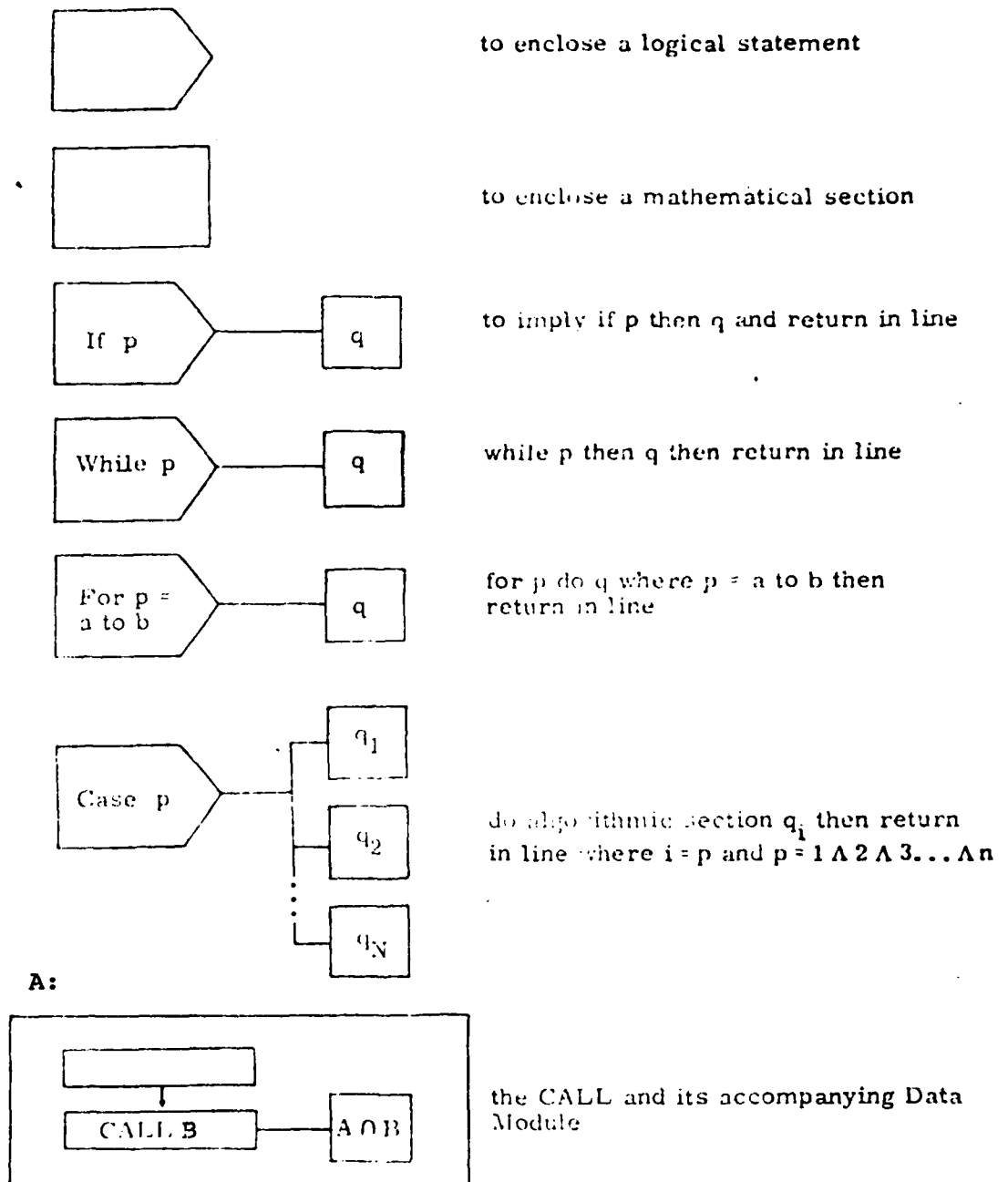


Figure 6.3.5.3
Design Diagram Notation

6.3.6 Interactive Debugger

An interactive debugging aid is another essential feature of the interim ISDS/HOS time-sharing environment. The debugger should be symbol-oriented rather than machine-oriented to allow references to variables by name. The debugger should allow the user to:

- suspend execution of his program (break).
- look at data or code.
- modify data or code.
- perform transfers.
- call procedures.
- trace the stack being used.
- look at procedure arguments.
- control and coordinate breaks.
- continue execution after a break.
- print machine registers.

Performing transfers and modifying code via the debugger is for debugging only and does not constitute a reliability issue for program development.

6.3.7 Interpreter

Certain missions, due to strategic requirements, place tight restrictions on the size, weight and power of embedded computer systems. As a result, additional requirements are placed on the software system.

By limiting the size of the target computers, common routines (such as trigonometric functions, exponential functions, matrix operations, etc.) might be precluded from

being placed more than once as in-line code. Instead these functions would be placed in a common subroutine library. For less tightly constrained systems, these subroutines would normally be linked into the main modules and accessed via CALL sequences. For more tightly constrained systems where sufficient space cannot be allocated for the CALL sequences, the common subroutines could be accessed via an interpreter.

A special in-line instruction invokes the interpreter. The interpreter then processes the next instruction in sequence to determine the next common routine to be processed and the location of any input data. Control is then passed to the desired common routine until completion, when the interpreter again takes over. The next instruction in the executing module is examined in a like manner until an in-line instruction indicates that control is to be returned to the executing module.

Through the use of an interpreter, the inclusion of common logic in in-line code (wherever that code is used) is eliminated at the expense of a slower execution speed, due to the system overhead incurred because of the interpreter.

The interpreter, as used in the interim ISDS/HOS environment, will be designed in accordance with the ISDS/HOS concepts and in particular, with the axioms of HOS. The unique feature of the interpreter is that it will ensure, in real time, interface correctness and reliability between the executing module and the common subroutines.

The use of common subroutines via an interpreter is a natural by-product of the HOS software-specification and design process. In particular, in an HOS functional decomposition or control map, any given node in the hierarchical tree structure is not concerned with who invoked it but only with completing its function. Upon completion of the given function, control

returns to the controlling node. In this sense, the interpreter is controlled by the executing function with the required data for the interpreter being the instruction stream of the executing module. The common subroutines are controlled by the interpreter.

In ISDS/HOS the common subroutines could be considered as primitive operations similar, for example, to an add instruction. If the primitive operations were hardware, they could be executed outside of the scope of the software. The use of these primitive operations could result in a significant savings in execution time over an interpreter, but at the expense of added cost to the hardware.

7.0 CONCLUSION

7.0 CONCLUSION

The Integrated Software Development System/Higher Order Software has been presented as a formalized approach to the design and development of reliable cost-effective computer-based systems. Higher Order Software (HOS) is the formal systems theory that has provided the foundation of ISDS/HOS. The methodology of ISDS/HOS can be used to explicitly describe all systems including hardware, firmware, software, and humanware, as well as the dynamic environment within which these systems may reside. A system specification was defined to be an abstract hierarchical decomposition depicting the functional characteristics of a system. A "function" has been described as a specific transformation from a particular input set to its related output set. A function, its input set, and its output set comprise a system module which is hierarchically decomposed into component sub-modules according to a formal set of axioms. Functional decompositions that comply with the axioms of ISDS/HOS are guaranteed to have consistent modular interfaces as well as explicitly defined functional control, where control is a formally specified affect of one module to another. Such an axiomatic theory is unique to the ISDS/HOS methodology.

ISDS/HOS as described is comprised of the HOS theory, a complete range of software development tools, and a system of standards for developing computer-based systems. The principles of ISDS/HOS are applied to all phases of system development throughout all disciplines including design, implementation, documentation, and management. The key features of ISDS/HOS that were outlined include the standard management procedures, static verification, flexibility in systems, and automated tools.

An important management procedure developed for ISDS/HOS is the Assembly Control Supervisor (ACS) concept. This method establishes a focal point through which all official modules are filtered, thereby providing increased management visibility and systems integrity.

Static verification assures that an ISDS/HOS specification is consistent with the axioms. This can be done automatically, without execution of the system, through use of the Design Analyzer Tool, which statically checks the software interfaces.

Flexible systems are necessary to allow smooth adaptability to specification or requirement changes throughout the system development. With the development of ISDS/HOS, a complete trace of all repercussions resulting from a component modification within the system is immediately available, automatically. Thus, system modification can be implemented with a minimum of effort and the elimination of all possible side effects is guaranteed.

It has been an integral concept of the ISDS/HOS methodology that the tools and techniques are used to define and describe all aspects of the system throughout all phases of development. Thus, common tools are used to define and describe the functions and interfaces, the execution flow, the verification processes, and the management processes of a system, providing a unified structure within which development can proceed in a standardized, tractable manner.

The objective of ISDS/HOS has been to provide a mechanism to tie together the entire system development process, incorporating engineering standards developed from formal foundations in order to eliminate many of the traditional sources of design and implementation errors.

BIBLIOGRAPHY

BIBLIOGRAPHY

- ASC75 Asch, A., Kelliher, D.W., Locher III, J.P., Connors, T.,
 "DoD Weapon Systems Software Acquisition and Management
 Study Volume 1, MITRE Findings and Recommendations,"
 Vol. 1, MTR-6908, May 1975.
- BRA75 Bratman, Harvey and Court, Terry, "The Software Factory",
 Computer, Vol. 8, No. 7, May 1975.
- BEN76 Bender, G., "Structured Design", Hughes Aircraft Corp.,
 Ground Systems Group, Fullerton, CA, 9 September 1976.
- BOE72 Boehm, B.W. and Hailey, A.C., et.al., "Information Pro-
 cessing/Data Automation Implications of Air Force
 Command and Control Requests in 1980, Executive Summary",
 Report SAMSO/XRS. 71.1, U.S. Air Force, 1972.
- CEN75 "CENTACS: Information Brochure", U.S. Army Center for
 Tactical Computer Sciences, October 1975.
- CHO57 Chomsky, N., Syntactic Structures, Mouton, The Hague,
 1957.
- DAH72 Dahl, O.J., Dijkstra, E.W., and Hoare, C.A.R., "Structured
 Programming", C.A.R. Hoare, Gen. Ed., New York, Academic,
 1972.
- DAM76 Damon, Evmenios, "Domonic", NASA/Goddard Spacecraft
 Center, Greenbelt, MD, 1976.
- DAV76 Davis, C.G. and Vick, C.R., "The Software Development
 System", in supplement to Proceedings for the 2nd
 International Conference on Software Engineering, San
 Francisco, CA, IEEE Catalog No. 76H1125-4 C, Oct. 1976.
- DDR74 DDR&E Memorandum for Assistant Secretaries of the Military
 Departments (R&D), "Computer Software", 20 March
 1974.
- DDR75 DDR&E Memorandum for Assistant Secretaries of the Military
 Departments (R&D), "DoD Software Technology Research
 Program", 14 October 1975.
- DER75 DeRoze, Barry, "An Introspective Analysis of DoD System
 Software Management", Defense Management Journal,
 October 1975.

- GAN76 Gansler, Jacques, S., "Remarks", before the Polytechnic Institute of New York, Microwave Research Institute, Symposium on Computer Software Engineering, NY, NY, News Release from Office of Assistant Secretary of Defense, Washington, D.C., April 20, 1976.
- GRI71 Gries, D., Compiler Construction for Digital Computers, Wiley and Sons, New York, 1971.
- GRI76 Gries, D., "An Illustration of Current Ideas on the Deviation of Correctness Proofs and Correct Programs", in Proceedings of the 2nd International Conference on Software Engineering, San Francisco, CA, IEEE Catalog No. 76H1125-4 C, October 1976.
- GO075 Goodenough, J.B., "Exception Handling: Issues and a Proposed Notation", Comm. ACM, Vol. 18, No. 12, December 1975.
- GUT75 Guttag, J., "The Specification and Application to Programming of Abstract Data Types", Univ. of Toronto Technical Report CSRG-59, Sept. 1975.
- HAM71 Hamilton, M., "Management of Apollo Programming and Its Application to the Shuttle", The Charles Stark Draper Laboratory, Cambridge, MA, Software Shuttle Memo #29, 1971.
- HAM72 _____, "The AGC Executive and Its Influence on Software Management", The Charles Stark Draper Laboratory, Cambridge, MA, Shuttle Management Note 2, February 1972.
- HAM73a _____, and Zeldin, S., "Higher Order Software Requirements", The Charles Stark Draper Laboratory, Cambridge, MA, Doc. E-2793, August 1973.
- HAM73b _____, and _____, "Higher Order Software Techniques Applied to a Space Shuttle Prototype Program", in Lecture Notes in Computer Science, Vol. 19, Goos and J. Harmanis, Eds., New York, Springer-Verlag, pp. 17-31, presented at Program Symp. Proc. Colloque sur la Programmation, Paris, France, August 1973.
- HAM73c _____, "Design of the GN&C Flight Software Specification", The Charles Stark Draper Laboratory, Cambridge, MA, Doc. C-3899, February 1973.
- HAM76a _____, and Zeldin, S., "Higher Order Software--A Methodology for Defining Software", IEEE Transactions in Software Engineering, Vol. SE-2, No. 1, March 1976.

- HAM76b _____, and _____, "The Foundations for AXES: A Specification Language Based on Completeness of Control", The Charles Stark Draper Laboratory, Cambridge, MA, Doc. R-964, March 1976.
- HAMP76 Hampton, N. and Myers, G., "System Design Laboratory", Naval Electronics Laboratory Center, San Diego, CA, September 1976.
- IBM IBM Corp., "Installation Management: HIPO--A Design Aid and Documentation Technique", GC20-1851-1.
- JAC76 The Jackson Design Methodology Handbook of Program Design, INFOTECH International, 1976.
- KOS75 Kossiakoff, A., Sleight, T.P., Prettyman, E.C., Park, J.M., and Hazan, P.L., "DoD Weapon Systems Software Management Study", Johns Hopkins University Applied Physics Laboratory, APL/JHU SR75-3, June 1975.
- LED75 Ledgard, F.F. and Marcotty, M., "A Geneology of Control Structures", Comm. ACM, Vol. 18, No. 11, November, 1975.
- LIE73 Lieblein, Edward, "Problems in Software Development", Keynote Address Presented to the Joint Services Electronics Program Topical Review in Information Services, University of Illinois, 16 October 1973.
- MAL75 Mallach, Efrem G., "Emulator Architecture", Computer, Vol. 3, No. 3, pp. 24-32, August 1975.
- MIL71 Mills, H.D., "Top-Down Programming in Large Systems", in Debugging Techniques in Large Systems, ed. R. Reistin, Prentice Hall, New Jersey, 1971.
- MUL72 Muller, E.S., "Shuttle Unified Navigation Filter", Space Shuttle GN&C Equation Document No. 21, The Charles Stark Draper Laboratory, Cambridge, MA, November, 1972.
- MYE74 Myers, G.J., Reliable Software Through Composite Design, Petrocelli, Mason and Charter, N.Y., 1975.
- RAM75 Ramamoorthy, C.V., and Ho, Siu-Vin F., "Testing Large Software with Automated Software Evaluation Systems", IEEE Transactions on Software Engineering, Vol. SE-1, No. 1, March 1975.

- R&D76 R&D Technology Panel, "Proposed Computer Resource Technology Objectives", as modified by DDR&E, July 12, 1976.
- ROS76 Ross, Donald T., and Schoman, Kenneth, E., "Structured Analysis for Requirements Definition" in supplement to Proceedings of the 2nd International Conference on Software Engineering, San Francisco, CA, IEEE Catalog No. 76H1125-4 C, October 1976.
- SHU66 Shubik, Martin, "Simulation of the Industry and the Firm", in Computer Simulation Techniques, Naylor, T., et.al., eds., Wiley and Sons, New York, 1966.
- SNO72 Snowden, R.A., "PEARL: An Interactive System for the Preparation and Validation of Structured Programs", Sigplan Notices, March 1972.
- STE74 Stevens, W.P., Myers, G.J., and Constantine, L.L., "Structured Design", IBM Systems Journal, Vol. 13, No. 2, 1974.
- STR76 Straeter, T., "MUST (Multipurpose User-Oriented Technology) Program Plan Preliminary", NASA/Langley Research Center, Hampton, VA, 21 October 1976.
- TEI76 Teichroew, Daniel and Hershey III, Ernest Allen, "Computer-Aided Structured Documentation and Analysis of Information Processing System Requirements", ISDOS Project, Department of Industrial and Operations Engineering, University of Michigan, Ann Arbor, MI, August 1976.
- WIR72 Wirth, N., Systematic Programming: An Introduction, Prentice-Hall, Inc., New Jersey, 1972.
- WU74 Wu, Y.S., et.al., "Report on the Navy Ad Hoc Software Maintainability Committee", August 12, 1974.
- YOU75 Yourdon, E., Techniques of Program Structure and Design, Prentice-Hall, Inc., New Jersey, 1975.

APPENDIX I
DEFINITIONS AND PROPERTIES OF CONTROL

APPENDIX I: DEFINITIONS AND PROPERTIES OF CONTROL

We consider a system to be a hierarchy in which the elements of the hierarchical system are the mathematical functions and the defining relation of that hierarchy is that of control. The elementary properties that we attribute to the notion of control are stated as the axioms of Higher Order Software (HOS) . The rationale for these axioms are based on experience and analysis of interface relationships associated with the development of large-scale, multiprogrammed systems.

The following symbols*:

\forall , for every	ϕ , controls
\wedge , logical 'and'	\nexists , does not control
\vee , logical 'or'	\nmid , interrupts
\in , element of	\nmid , does not interrupt
\notin , not an element of	\exists , there exists
$ $, such that	$!$, a unique
\subset , subset of	$=$, logical 'equals'
\cup , union of	\neq , logical 'does not equal'
\varnothing , empty set	$a \rightarrow b$, logical 'if a then b'

iff, if and only if

variables: e.g., x, y, x', y', A, B ; function names: e.g., A is a function name of $y=A(x)$ where y and x are access variables; and brackets: e.g., $\{ \}$ are used in the discussion below. In addition, the following notation is also used:

*For any symbol in which the negative is not specifically stated, a vertical line "|" or oblique line "/" is drawn through the symbol to indicate the opposite or negative meaning of that symbol.

x	A variable represents any element of a class of objects. In particular, we distinguish those variables whose values define the input space or output space of a function to be access variables. The same access variable can be used to represent more than one class of objects if each representation refers to a subclass of the same class.
$\{a,b\}$	class of two elements; i.e., $\{x x=a \vee x=b\}$
(a,b)	the ordered pair, i.e., the class $\{\{a\}, \{a,b\}\}$; alternately, the class $\{\{a,\varnothing\}, \{b,\{\varnothing\}\}\}$.
$x_{\{x P(x)\}}$	access variable x representing class of objects which satisfy $P(x)$ any variables referenced in $P(x)$ other than x are considered constant with respect to $P(x)$, e.g., $\{x x < z\}$
$(x_1, x_2 \dots x_A)$	an ordered A-tuple element. The A-tuple $(x_1, x_2 \dots x_A)$ is an ordered A-tuple in that it implies x_1 , as the first element, x_2 as the second element, x_A as the last element. Alternatively, for example, an ordered 4-tuple implies if $(a,b,c,d) = (e,f,g,h)$ then, $a=e$, $b=f$, $c=g$, $d=h$.
$\{(x_1, x_2 \dots x_A) (x_1, x_2 \dots x_A) \in Q\}$	set of ordered A-tuple elements
$(x_1, x_2 \dots x_A) [S]$	an ordered A-tuple referring to module S
$x[S]$	an access variable referring to module S

$$X = \{x_1, x_2, \dots, x_A\}$$

alternatively,

$$x_j^{\{j|j=1,2,\dots,A\}} = (x_1, x_2, \dots, x_A)$$

set X of A ordered Access variables, i.e., $X = \{x | x = x_1 \vee x = x_2 \vee \dots \vee x = x_A\}$

$$X_{F,G}$$

hierarchical element identification (see page AI-6) where the hierarchical element is a set of ordered access variables.

$$\{X_{F,G}\} = X_F^{\{F|F=1,2,\dots,N\}} G$$

alternatively,

$$\{X_{F,G}\} = \{X_{1,G}, X_{2,G}, \dots, X_{N,G}\}$$

set of hierarchical elements at a given level of control (see page AI-6)

$$(x_1^{\{x_1|P_1(x_1)\}} ,$$

$$x_2^{\{x_2|P_2(x_2)\}} \dots$$

$$x_A^{\{x_A|P_A(x_A)\}})$$

an ordered A -tuple element of access variables where:

$\{x_1|P_1(x_1)\}$ is the class that x_1 represents; and,

$\{x_2|P_2(x_2)\}$ is the class that x_2 represents; and,

$\{x_A|P_A(x_A)\}$ is the class that x_A represents.

The following definitions are provided as an aid to understanding the aspects of control.

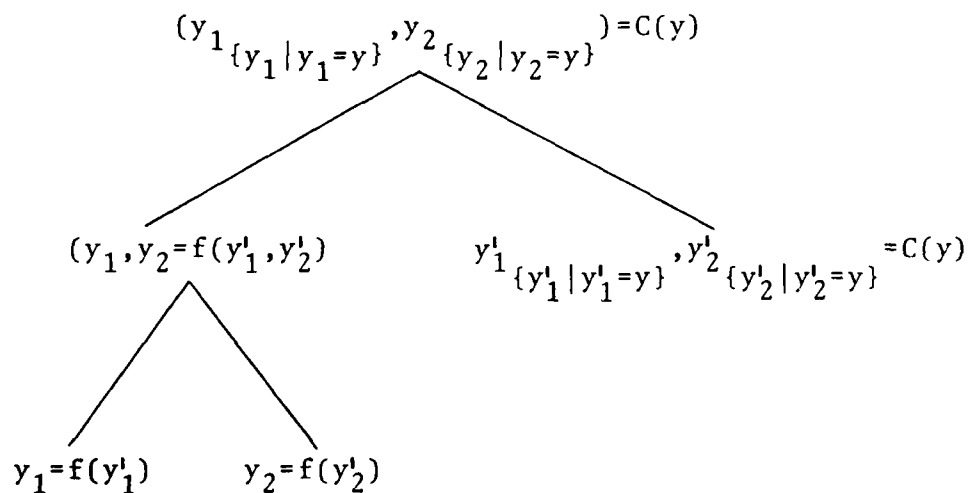
We define an A-dimensional input space by the values of the A variables $(x_1, x_2 \dots x_A)$. And we define a B-dimensional output space by the values of the B variables, $(y_1, y_2 \dots y_B)$. An element of the output set, $p \in P$, is a particular point for $(y_1, y_2 \dots y_B)$.

A function, $F: Q \rightarrow P$ or $P = F(Q)$, is a mapping from the input set, Q, to the output set, P. Each element of the input set is expressed as a unique element of the output set. That is, the domain of the function is Q and the range of the function is P such that $F = \{(q, p) \mid \forall q \in Q \exists! p, p \in P\}$. $p = F(q)$ has the same meaning as $(q, p) \in F$ whereas $P = F(Q)$ has the same meaning as F. When we refer to an element in the hierarchy other than the topmost, or root, element, the function, F, is defined with reference to the particular access variables associated with that element.

We define a controller, the module, to be a collection of mathematical functions whose interface properties uniquely distinguish each element of the collection. If, for example, A and B are modules; then, $A \circ B$ is read "A control the invocation of B." We say B is a function of A. The symbol for control is also used when describing a module with respect to other aspects of the relationship between the module and the hierarchical elements it controls. For example, if A is a module and X is a set of input variables of B, then $A \circ X$ is read "A controls the access rights for the set of input variables, X, of B."

The module exists at the node just immediately higher on the tree (or hierarchy) relative to the functions it controls. Each node (any point at which two or more branches intersect) of the tree represents a unique point of execution of a function. Each node and all of its dependents represent the unique tree structure, T.

When the choice of subfunctions is limited so that at least one subfunction implies the same total set of ordered pairs as that of the controller, we do not decompose that function. For example, in Figure AI-1, C cannot be decomposed.



In the above case, function C is primitive. Therefore, C controls the empty set with respect to function, input variables, output variables and dependent trees.

DEFINITION: *The formal control system (Figure AI-2) is one in which each module, S , has a unique identification*

$$S_{n_i, m_i} \equiv [P_{n_i, m_i} = F_{n_i, m_i}(Q_{n_i, m_i})]$$

n_i, m_i defines a particular level of control in which i is the nested level of the module. $i=1$ implies the level directly below the top node (or root of the tree). At each level, there is a set, N_i , of N node positions such that $N_i = \{1, 2, \dots, N\}$ and $n_i \in N_i$. n_i is the node position (from the left) relative to its most immediate higher node, m_i . m_i is the recursive relationship $m_i = n_{i-1}, m_{i-1}$ defined for $i > 2$. If $i=2$; $m_i = n_{i-1}$. If $i=1$; $n_i, m_i = n_i, \{\varnothing\}$. If $i=0$; $n_i, m_i = \{\varnothing\}, \{\varnothing\}$. *

*Henceforth, n_i, m_i will be represented as n_i, m_i , but should be interpreted as n_i, m_i .

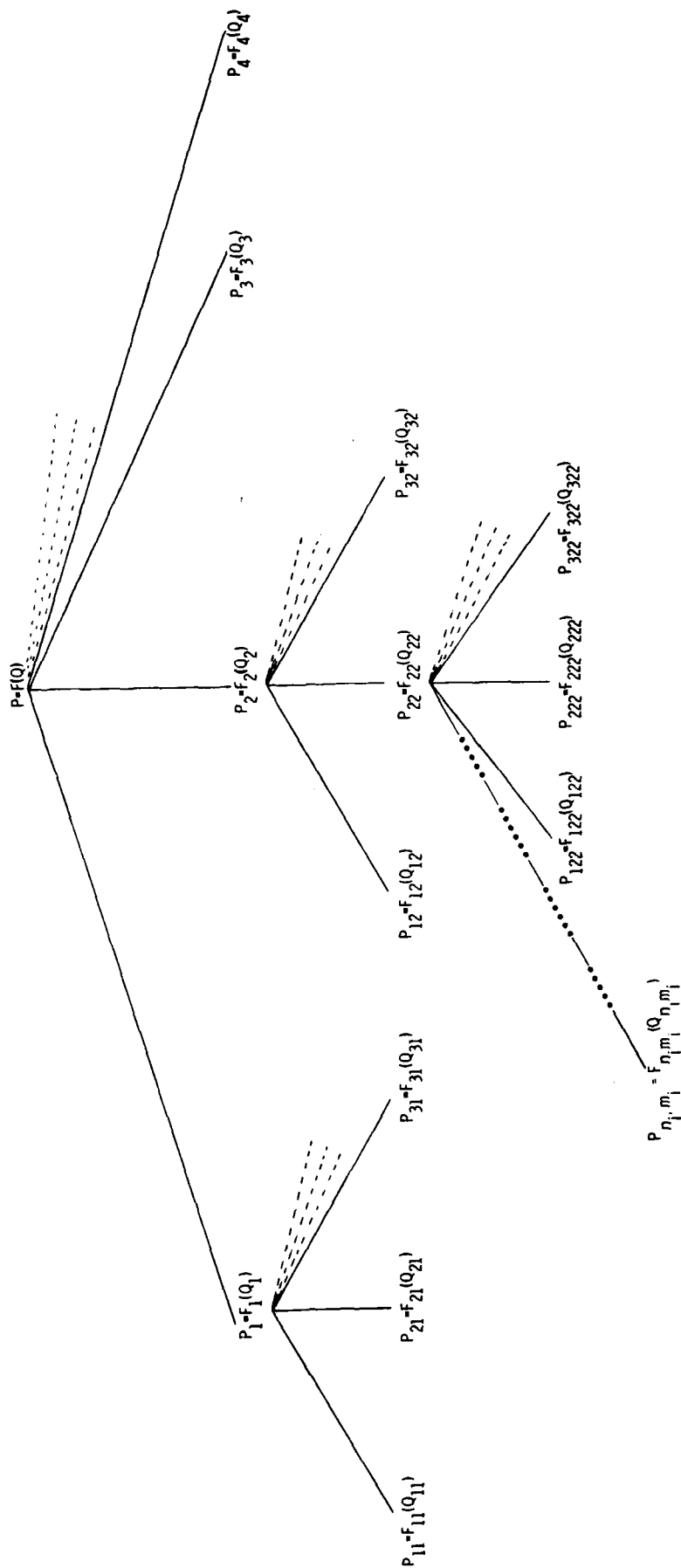


FIGURE AI-2

THE FORMAL CONTROL SYSTEM: $S_{n_1, m_1} = F_{n_1, m_1}(Q_{n_1, m_1})$

DEFINITION: Invocation provides for the ability to perform a function. We perform a function when a particular element of the input set produces the corresponding element of the output set. A function within the hierarchy is valid if it contributes in some way to the performance of the controller's function. If A and B are modules, we say

$$A \text{ invokes } B \text{ iff } (q,p) [A] \rightarrow (q,p) [B]$$

axiom 1: The module, $S_{n_i m_i}$, controls the invocation of the set of valid functions on its immediate, and only its immediate, lower level, $\{F_{n_{i+1} n_i m_i}\}$. That is:

$$\forall j \forall n_{i+1} \in N_{i+1} \exists! S_{n_i m_i} \cdot [(S_{n_i m_i} \circ F_{n_{i+1} n_i m_i}) \wedge ((n_j m_j \neq n_{i+1} n_i m_i) \rightarrow S_{n_i m_i} \not\circ F_{n_j m_j})]$$

Thus, the module, $S_{n_i m_i}$, cannot control the invocation of functions on its own level.

It also follows that the module, $S_{n_i m_i}$, cannot control the invocation of its own function.

DEFINITION:

Responsibility - If A is a module, A is responsible for elements of the output set if for every given element of the input set

$$q[A] \rightarrow P[A]$$

axiom 2: The module, $S_{n_i m_i}$, controls the responsibility for elements of the output space, of only $P_{n_i m_i}$, such that the mapping $F_{n_i m_i}(Q_{n_i m_i})$ is $P_{n_i m_i}$. That is:

$$\forall j \forall n_i m_i \exists! S_{n_i m_i} \cdot ((S_{n_i m_i} \circ P_{n_i m_i}) \wedge ((n_j m_j \neq n_i m_i) \rightarrow S_{n_i m_i} \notin P_{n_j m_j}))$$

Thus, there must not exist any member of the input space for which no member of the output space is assigned. For, if this were not the case, we would have an invalid function.

DEFINITION: An access right provides for the ability to locate an element of a given set of variables, and once located, the ability to reference or replace a value of said element. If $v_1, v_2 \dots v_n$ are values associated with access variable w , i.e., $w\{w | w \in \{v_1, v_2 \dots v_n\}\}$ and V represents a class of two elements such that v_c is established as the "chosen" element of the n elements of w

$$V = \{v_c, \{v_c, \{v_1, v_2 \dots v_n - v_c\}\}\}$$

Then, if A and B are modules, A and B have access rights to variable w iff

$$w[A] \rightarrow v_c \wedge w[B] \rightarrow v_c$$

axiom 3: The module, $S_{n_i m_i}$, controls the access rights to each set of variables, $\{Y_{n_{i+1} n_i m_i}\}$, whose values define the elements of the output space for each immediate, and only each immediate, lower level function.

$$\forall j \forall n_{i+1} \in N_{i+1} \exists! S_{n_i m_i} [(S_{n_i m_i} \circ Y_{n_{i+1} n_i m_i}) \wedge ((n_j m_j \neq n_{i+1} n_i m_i) \rightarrow S_{n_i m_i} \neq Y_{n_j m_j})]$$

NOTE: If any two modules, $S_{n_i m_i}$ and $S_{n_j m_j}$, require the same function formulation, the same set of computer residing instructions can be used for the functions as long as the access rights of the variables are controlled via axiom 3.

axiom 4: The module, $S_{n_i m_i}$, controls the access rights to each set of variables, $\{X_{n_{i+1} n_i m_i}\}$, whose values define the elements of the input space for each immediate, and only each immediate, lower level function.

$$\forall j \forall n_{i+1} \in N_{i+1} \exists! S_{n_i m_i} \cdot \{ (S_{n_i m_i} \circ X_{n_{i+1} n_i m_i}) \wedge ((n_j m_j \neq n_{i+1} n_i m_i) \rightarrow S_{n_i m_i} \neq X_{n_j m_j}) \}$$

Thus, the module, $S_{n_i m_i}$, cannot alter the members of its own input set, i.e., the access to the elements of the input set of $S_{n_i m_i}$ cannot be controlled by $S_{n_i m_i}$.

DEFINITION:

Rejection - If A and B are modules, A rejects invalid input elements of A iff

$$q_{[A]} = \varphi \rightarrow p_{[A]} = \varphi; \text{ or,}$$

$$q_{[A]} \rightarrow (p, q)_{[B]} = \{\{\varphi\}, \{\varphi, \varphi\}\} \rightarrow p_{[A]} = \varphi$$

axiom 5: The module, $S_{n_i m_i}$, controls the rejection of invalid elements of its own, and only its own, input set, $Q_{n_i m_i}$. That is:

$$\forall j \forall n_i m_i \exists! S_{n_i m_i} \cdot \{ (S_{n_i m_i} \circ Q_{n_i m_i}) \wedge ((n_j m_j \neq n_i m_i) \rightarrow S_{n_i m_i} \neq Q_{n_j m_j}) \}$$

DEFINITION: Ordering provides for the ability to establish a relation in a set of functions so that any two function elements are comparable in that one of said elements precedes the other said element. If any two elements in $\{F\}$ are comparable with respect to relation, R , this implies

$$(F, F) \in R \quad \forall F \in F$$

$$(F_1, F_2) \in R \wedge (F_2, F_1) \in R \rightarrow F_1 = F_2$$

$$(F_1, F_2) \in R \wedge (F_2, F_3) \in R \rightarrow (F_1, F_3) \in R$$

$\{F_1, F_2, F_3\}$ is well-ordered.

axiom 6: The module, $S_{n_i m_i}$, controls the ordering of each tree, $\{T_{n_{i+1} n_i m_i}\}$, for the immediate, and only on the immediate, lower level.

$$\forall j \quad \forall n_{i+1} \in N_{i+1} \quad \exists! S_{n_i m_i} \cdot [(S_{n_i m_i} \circ T_{n_{i+1} n_i m_i}) \wedge ((n_j m_j \neq n_{i+1} n_i m_i) \rightarrow S_{n_i m_i} \phi T_{n_j m_j})]$$

Thus, the module, $S_{n_i m_i}$, controls the ordering of the functions, the input set and the output set, for each node of $\{T_{n_{i+1} n_i m_i}\}$.

APPENDIX II
DERIVATION OF THE PRIMITIVE CONTROL STRUCTURES

APPENDIX II: DERIVATION OF THE PRIMITIVE CONTROL STRUCTURES

Proper decomposition implies that each subfunction is necessary and that the set of subfunctions is sufficient so that the controller function is established in an unambiguous manner (cf. Appendix I).

When a node of a control map (c.f. Appendix I) is to be decomposed into subfunctions, we consider the input data type X , the output data type Y , the input set Q , the output set P , the data structure for X , $\{x_1, x_2 \dots x_A\}$, and the data structure for Y , $\{y_1, y_2 \dots y_B\}$.

Consider a particular input set, Q , and the corresponding set of input variables, X . We can represent the input set, Q , as a set of ordered A -tuples

$$\{(x_1, x_2 \dots x_A) \mid (x_1, x_2 \dots x_A) \in Q\} \quad (1)$$

Alternatively, we can represent the ordered A -tuple itself as a hierarchy of ordered pairs

$$(x_1, x_2 \dots x_A) = \{\{x_1\}, \{x_1, (x_2 \dots x_A)\}\} \quad (2)$$

Using an alternate formulation for ordered pairs such as

$$\{\{x_1, \varnothing\}, \{(x_2 \dots x_A), \{\varnothing\}\}\} \quad (3)$$

leads to an abbreviated form for an ordered set of variables $\{x_1, x_2 \dots x_A\}$ where x_1 is first, x_2 second, x_A is last.

In a similar manner, we can represent a particular output set, P , and its corresponding set of output variables, Y . As a data type, we refer to the set of input variables as X . As a data

structure, we refer to the set of variables $\{x_1, x_2 \dots x_A\}$ as a data structure. When we refer to the data type, X , we partition elements of Q for decomposition. When we refer to the data structure, X , we partition elements of X for decomposition.

The derivation of primitive control structures is based on
a) decomposition as a complete formulation of a function; and,
b) preliminary properties of control structures including single assignment and single reference. The proof is then based on showing the validity of a decomposition level with respect to its controller. Those control structures that are derived from combinations of X and Y represented by (1) or (3), and shown to be consistent with the HOS axioms, are defined as the primitive control structures.

Decomposition as a Complete Formulation of a Function

We establish the necessity of investigating X and Y of one control node represented by (1) or (3) at the subfunction level.

Suppose the data type X is not represented at the next most immediate lower level of decomposition as X , as Q distributed among the subfunctions or as a data structure for X distributed among the subfunctions. Examples of such a possibility are seen in Figure AII-1.

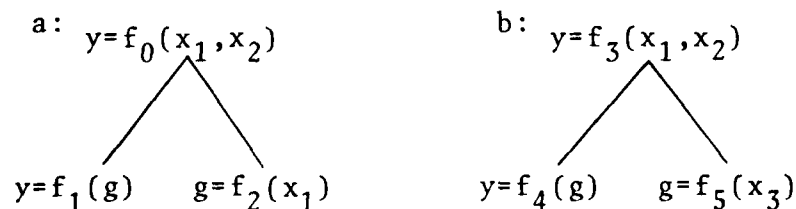


FIGURE AII-1

AII-2

From the definition of invocation and the property of control of invocation (axiom 1, Appendix I), a value of the input set of a controller must influence or contribute to the values of the input set of its subfunctions. In addition, a module must control access rights to the variables of a subfunction (from axiom 3 and axiom 4). This implies that if a variable is used by a controller and by a subfunction, the same value is used by both the controller and its subfunctions. In Figure AII-1a, we violate axiom 1 in that for various x_2 values, y always has the same constant value for a particular value of x_1 . This means that if the mapping were different for f_0 of Figure AII-1a due to different values of x_2 , the correct mapping could not be expressed without x_2 appearing in a subfunction. In Figure AII-1b, we violate axiom 4 because there is no way to establish the access rights to f_5 from controller f_3 .

Likewise, a similar deduction can be established for output variables and values, for there must be a mechanism to access an element of the output set. Thus, to conform to the axioms of HOS, we must achieve decomposition as a complete formulation of a function. To do this, we must investigate representations of X and Y of a controller as somehow being maintained at the subfunction level by data types, data structures, and sets of values.

Update Functions

It is helpful to establish the proof that a given variable cannot be both referenced and assigned by the same function before establishing the proof of the primitive HOS control structure.

Lemma 1 and Lemma 2 are used to show the proof of Theorem 3,4.1 which establishes the no update property of an HOS system.

Lemma 1: If x is a variable of a tuple of the input set at level m_i , then x cannot be a variable of a tuple of the output set at level $n_i m_i$.

- a) By the definition of decomposition as a complete formulation of a function, a variable of the input set of a controller must be accessed as an input variable of a subfunction.
- b) Consider f_0, f_1, f_2 as seen in Figure AII-2. The module, S_0 , corresponding to function f_0 does not control access to x because access to x via f_1 is not unique (i.e., there are two ways for f_1 to get x). This is in direct violation of axiom 3 and axiom 4.

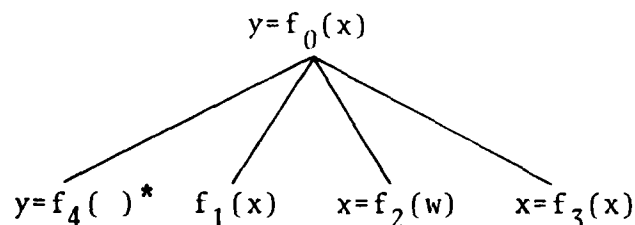


FIGURE AII-2

If f_0 gave f_1 access to get x from the output set of f_2 , we get an invalid function for f_0 . This is in violation of axiom 1. For no matter what element of x is input to f_0 , that input is ignored.

If we do not use f_2 , f_2 is extraneous and must be removed via Theorem 1.2

Thus, we have shown that if x is referenced by a given controller, then x cannot be referenced and assigned by different subfunctions of that controller.

*The brackets do not imply the empty set; the notation is used only for convenience of illustration.

- c) Consider f_0, f_3, f_4 as seen in Figure AII-2. If f_3 assigns x , $S_0 \not\vdash x$ because, again, there are two ways for f_3 to get x .

If f_0 gave access to f_3 to use an element of its output set to obtain its own input element, we get an invalid function for f_0 . For again, the original input element for f_0 is ignored.

* If f_0 provides access to f_3 for x , x would have to be an input variable to another function such as f_4 or be extraneous.

If x is used by f_4 , then there are two ways for f_4 to get x and again, $S_0 \not\vdash x$.

Thus, we have shown that if x is referenced by a given controller, then x cannot be referenced and assigned by the same subfunction.

Conclusion: Since we have established that a variable of the input set of a controller must be referenced by one of its subfunctions (via a) and that input variables cannot be referenced and assigned by different subfunctions (via b) and that input variables cannot be referenced and assigned by the same subfunction (via c), then we establish that x cannot be assigned by a subfunction if it is referenced by its controller.

Lemma 2: If y is a variable of a tuple of the output set at level m_i , then y cannot be a variable of a tuple of the input set at level $n_i m_i$. Proof is similar to above.

Theorem 3,4.1 (revisited from (HAM73b)): A variable of the output set of a function cannot be a variable of a tuple of the input set of that same function.

Consider Figure AII-3. Here, ℓ is a local variable in

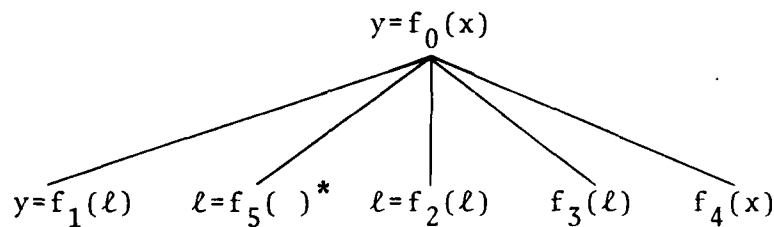


FIGURE AII-3

that it does not appear in the input or output of the controller f_0 . If ℓ is used as input to two functions (c.f. f_1 and f_3) or as output of two functions (c.f. f_5 and f_2) or as an update function (c.f. f_2) used in combination with f_1 or f_3 , the ordering of the subfunctions is not unique and the module corresponding to f_0 does not control the ordering of its subfunctions. This is in direct violation of axiom 6. Using this consideration, Lemma 1, Lemma 2, the proof of Theorem 3,4.1 can be shown.

Mutually Dependent Functions

Mutually dependent functions are those functions in which the output set of one function is the input set of the second function, and the output set of the second function is the input set of the first function.** We use the results of the following theorem as an aid in establishing the proof of primitive control structures.

* See footnote page AII-4

** Mutually dependent functions can be specified via the single assignment approach using recursive operations and different variables (c.f., discussion on single assignment which follows).

Theorem: If two variables are local to a given control level then, mutually dependent functions cannot be specified at the same level using the same two variables.

Such a situation is represented in Figure AII-4. Here,

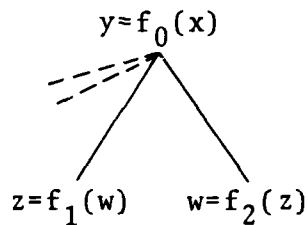


FIGURE AII-4

the order of f_1 with respect to f_2 is not unique. This is in violation of axiom 6. The module corresponding to function f_0 does not control the ordering of the dependent trees corresponding to functions f_1 and f_2 .

Single Assignment Property

The no update property of a control structure, the constraints imposed on mutually dependent functions, and the restriction to single-valued functions (c.f. Appendix I) imply that a variable is assigned only once per function performance. This property, single assignment, will also be used to derive the primitive control structures.

Single Reference Property

Single reference implies that an input variable of a controller node is accessed as an input variable of a subfunction only once per function performance. This restriction is based on the controlled ordering imposed on a set of subfunctions via axiom 6. Although ordering with respect to invocation can often

be ascertained from the data flow relationships, it is not always apparent that ordering with respect to access time or storage access or execution can be specified without the single reference approach. Such ordering relationships must be defined by a controller node. For example, consider Figure AII-5. Suppose f_1 completed the calculation of h and f_2 is initiated.

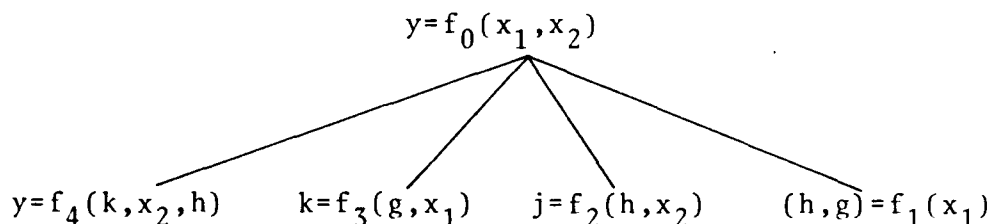


FIGURE AII-5: Implications of No Single Reference Property

When g is calculated, can f_3 interrupt f_2 if these functions are performed in a restricted implementation such as multi-programming? The answer is not apparent from data flow considerations. Since an arbitrary decision is required to approach the ordering relationship, the specification of f_0 is incorrect. Suppose, on the other hand, that f_3 and f_2 have been completed. Is the ordering with respect to storage access defined totally? Again, without a single reference property as seen in Figure AII-5, we need an arbitrary decision as to saving space for variables h and g (we do not know if h and g will be used in future calculations). Suppose, for another example, the inputs to f_3 are destroyed by some outer system problem. How can we determine how far to backtrack? Again, the single reference property would supply the proper information to each function in order to totally specify the backtrack ordering.

In order to specify total- or well-ordered relations among sub-functions, we must impose the single reference property on the input variable of a given set of subfunctions.

PRIMITIVE CONTROL STRUCTURES FOR HOS

In the following proof, we can assume, from the previous discussion, that an HOS control structure has at least the following properties:

- a) no update functions using the same variables
- b) no mutually dependent functions
- c) single assignment property
- d) single reference property
- e) the elements of the input set of the controller, and the elements of the output set of the controller are to be maintained in some form by its subfunctions
- f) at least two subfunctions are necessary to decompose a function

The proof that a primitive control structure is valid for an HOS system rests partially on the definition of a primitive control structure. Such a definition is derived from the concept of a construct class.

A construct class represents a function decomposition, the subfunctions of which can only be regrouped recursively. Figure AII-6 illustrates two possible regroupings of the same function. Note that the regrouped subfunctions do not change in any way. The most primitive level of a construct class decomposition is a primitive control structure. It is always possible to regroup the nonprimitive level as a nested hierarchy of primitive control structures.

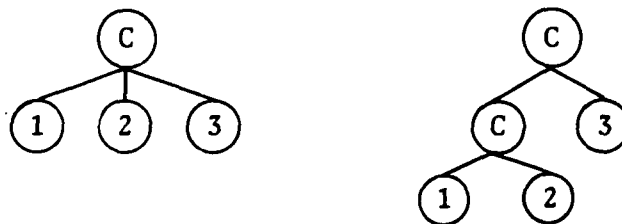


FIGURE AII-6: Recursive Regrouping of C

Case 1:

$$X_{A\{A|A \in N_i\}, m_i} = X_{m_i} ;$$

$$Q_{A\{A|A \in N_i\}, m_i} = Q_{m_i} ;$$

$$Y_{B\{B|B \in N_i\}, m_i} = Y_{m_i} ;$$

$$P_{B\{B|B \in N_i\}, m_i} = P_{m_i} ; \text{ and, } A \neq B$$

In this case, X_{m_i} appears directly in one subfunction at level i , relative to node m_i , and Y_{m_i} appears directly in another subfunction at level i , relative to node m_i . By eliminating the combinations assumed invalid from the control structure properties listed above, we are left with the following considerations.

Consider Figure AII-7. If Case 1 is valid, we must assume that: 1) x is input to one and only one subfunction; and, 2) a local variable is always output of one and only one subfunction and input to one and only one other subfunction.

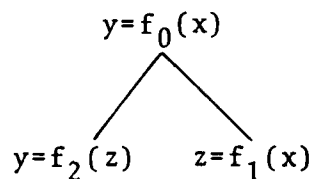


FIGURE AII-7: Composition as a Primitive Control Structure

Figure AII-7 is considered to be the primitive control structure, composition. The canonical form of this primitive control structure has two subfunctions as seen in Figure AII-7. Access to x and y are provided unambiguously in compliance with axiom 3 and axiom 4. In addition, the ordering of the subfunctions is unique in compliance with axiom 6. Each element of the controller's function is maintained by the subfunctions in compliance with axiom 1, axiom 2, and axiom 5.

Case 2:

$$(X_{A\{A|A \in N_i\}, m_i} \xrightarrow{\subset} X_{m_i}) \rightarrow (X_{C\{C|C \in N_i\}, m_i} = X_{m_i}) \rightarrow$$

$$(Q_{C\{C|C \in N_i\}, m_i} = Q_{m_i});$$

$$(Y_{B\{B|B \in N_i\}, m_i} \xrightarrow{\subset} Y_{m_i}) \rightarrow (Y_{C\{C|C \in N_i\}, m_i} = Y_{m_i}) \rightarrow$$

$$(P_{C\{C|C \in N_i\}, m_i} = P_{m_i}); \text{ and,}$$

$$A \neq B$$

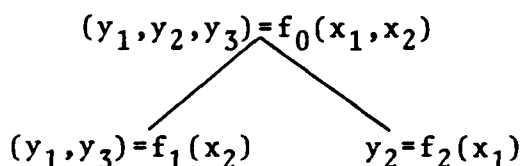


FIGURE AII-8

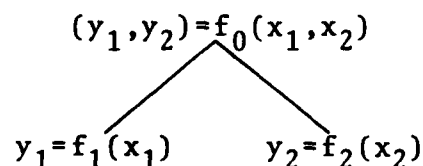


FIGURE AII-9: Class Partition
as a Primitive Control
Structure.

Consider Figures AII-8 & 9: In each case, by performing both subfunctions once, we obtain an element for (x_1, x_2) and (y_1, y_2) . But, consider Figure AII-8. The order of the ordered pairs is not unique in that the order for (x_1, x_2) conflicts with the order for (y_1, y_2, y_3) . Example: Suppose f_1 and f_2 had to contend for the same resources. Ultimately, there is no way to uniquely determine the ordering of these functions. However, if we restrict such a structure as in Figure AII-9, in which a one-to-one correspondence exists between the ordered (x_1, x_2) and the ordered (y_1, y_2) , then the "first" subfunction can always be ascertained from the order associated with the input and output variables of the controller.

Thus, Case 2 is a valid primitive control structure under the restriction that if f_1 is first, x_1 is first and y_1 is first such that $y_1 = f_1(x_1)$ and if f_2 is second, x_2 is second and y_2 is second such that $y_2 = f_2(x_2)$. Such an ordering is established by the controller and maintained by the subfunctions (Figure AII-9).

Case 3:

$$X_{A\{A|A \in N_i\}, m_i} = X_{m_i} ;$$

$$Q_{A\{A|A \in N_i\}, m_i} \subsetneq Q_{m_i} ;$$

$$Y_{B\{B|B \in N_i\}, m_i} = Y_{m_i} ;$$

$$P_{B\{B|B \in N_i\}, m_i} = P_{m_i} ; \text{ and, } A \nmid B$$

Case 4:

$$X_{A\{A|A \in N_i\}, m_i} = X_{m_i} ;$$

$$Q_{A\{A|A \in N_i\}, m_i} \subsetneq Q_{m_i}$$

$$Y_{B\{B|B \in N_i\}, m_i} = Y_{m_i} ;$$

$$P_{B\{B|B \in N_i\}, m_i} \subsetneq P_{m_i} ; \text{ and, } A \nmid B$$

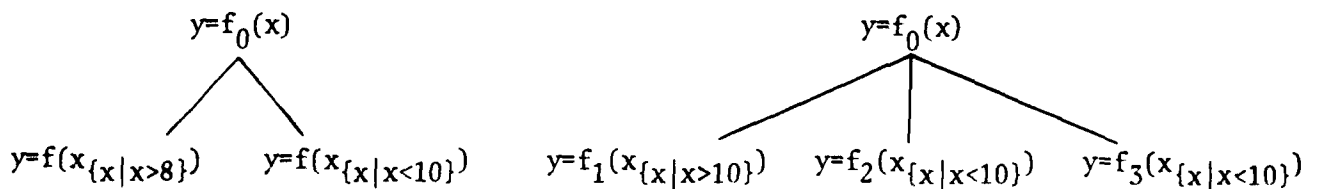


FIGURE AII-10

In Case 7 and Case 8, variable x does not appear directly in the subfunction. Due to the function definition for a control node, there is no general mechanism to restrict $P_{B\{B|B \in N_i\}, m_i} \neq P_{m_i}$.

Thus, we refer to all of Y_{m_i} in each subfunction and can then consider Case 7 and Case 8 together.

Consider Figure AII-10: In Figure AII-10a, we see that if x is not partitioned, there is no way to provide a unique ordering of the subfunctions in those cases where the element of the input set appears in both subfunctions. In addition, the partition is restricted so that the input variables of the subfunction cannot represent the same set of elements (c.f., Figure AII-10b). Again, a unique ordering of subfunctions is not possible under conditions such as presented in Figure AII-10b.

It follows that Case 7 and Case 8 are valid primitive control structures under the restriction that Q is divided into a partition, C , of Q such that $Q = C_j$, $c_j \subset C$, and $Q = \bigcup_i c_i$ and for two sets $c_j \cap c_i = \emptyset$.

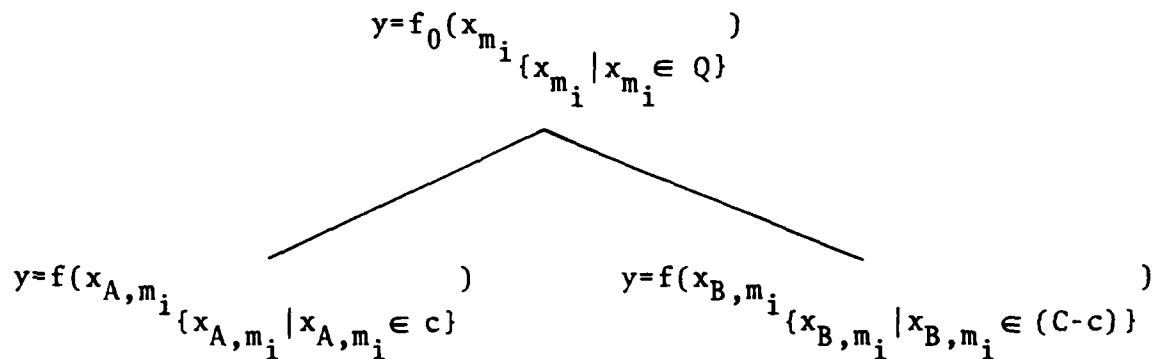


FIGURE AII-11

Set Partition Primitive Control Structure ($C = Q$ and $A \neq B$ where $A \in N_j$ and $B \in N_j$)

The canonical form of the primitive control structure has two sub-functions to restrict f_0 from controlling itself. Access to a particular element of the input set is unambiguous. Statically, the ordering of subfunctions is equal. Dynamically, the ordering is uniquely determined by that subfunction which has the element of Q chosen at a higher level.

Case 5:

$$x_{A\{A|A \in N_i\}, m_i} = x_{m_i} ;$$

$$Q_{A\{A|A \in N_i\}, m_i} = Q_{m_i} ;$$

$$Y_{B\{B|B \in N_i\}, m_i} = Y_{m_i} ;$$

$$P_{B\{B|B \in N_i\}, m_i} \subsetneq P_{m_i}^* ; \text{ and, } A \neq B$$

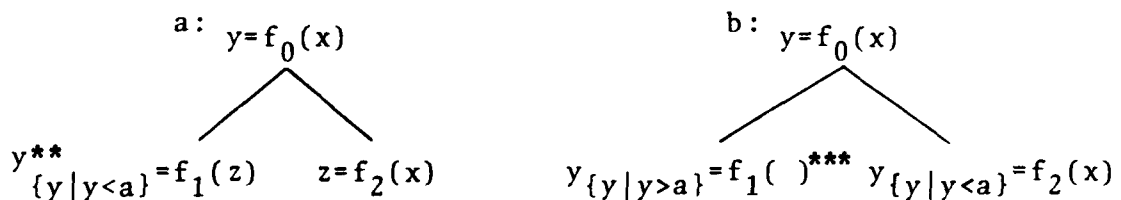


FIGURE AII-12

(a) Consider Figure AII-12 To account for all of Y_{m_i} elements, we need at least two subfunctions that assign $Y_{n_i m_i}$ elements.

* where $P_{B\{B|B \in N_i\}, m_i} \subsetneq P_{m_i}$ means $P_{B\{B|B \in N_i\}} \subset P_{m_i} \wedge P_{B\{B|B \in N_i\}} \neq P_{m_i}$

** In general, this can be any relation, $R(y)$.

*** See footnote page AII-4.

(b) Consider Figure AII-12b to respond to the requirement of Case (a): If f_1 has x as the input variable, we have violated the single assignment property (here such a violation results in a multivalued function). If f_1 has the output variable of f_2 as input, we violate the no update property (likewise, a violation of this nature also would result in a multivalued function). Case 5 is invalid for HOS, and thus, cannot be considered a primitive control structure.

Case 6:

$$X_{A\{A|A \in N_i\}, m_i} = X_{m_i};$$

$$Q_{A\{A|A \in N_i\}, m_i} = Q_{m_i};$$

$$(Y_{B\{B|B \in N_i\}, m_i} \stackrel{C}{\neq} Y_{m_i}) \rightarrow (Y_{C\{C|C \in N_i\}, m_i} = Y_{m_i}) \rightarrow$$

$$(P_{C\{C|C \in N_i\}, m_i} = P_{m_i}); \text{ and,}$$

$$A \neq B$$

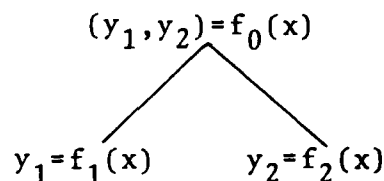


FIGURE AII-13

Consider Figure AII-13: The case remaining (after eliminating those control structures that violate the general properties

of control structures) as seen in Figure AII-13. Case 6 proves to be invalid by the violation of the single reference property. (Here, the violation implies an implicit data lock for x in order for f_1 and f_2 to attempt to access x concurrently.) Case 6 is invalid for HOS, and thus, cannot be considered a primitive control structure.

Case 7:

$$Y_{B\{B|B \in N_i\}, m_i} = Y_{m_i} ;$$

$$P_{B\{B|B \in N_i\}, m_i} = P_{m_i} ;$$

$$(X_{A\{A|A \in N_i\}, m_i} \overset{C}{\neq} X_{m_i}) \rightarrow (X_{C\{C|C \in N_i\}, m_i} = X_{m_i}) \rightarrow$$

$$(Q_{C\{C|C \in N_i\}, m_i} = Q_{m_i}); \text{ and,}$$

$$A \neq B$$

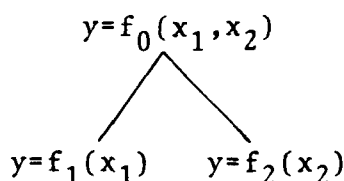


FIGURE AII-14

- (a) Consider Figure AII-14: In this case, to get one element for (x_1, x_2) , both subfunctions must be performed. If one subfunction assigns all y elements, then the other subfunction is either extraneous or invalid. Case 7 is invalid for HOS, and thus, cannot be considered a primitive control structure.

Case 8:

$$Y_{B\{B|B \in N_i\}, m_i} = Y_{m_i} ;$$

$$P_{B\{B|B \in N_i\}, m_i} \not\subseteq P_{m_i} ;$$

$$(X_{A\{A|A \in N_i\}, m_i} \not\subseteq X_{m_i}) \rightarrow (X_{C\{C|C \in N_i\}, m_i} = X_{m_i}) \rightarrow$$

$$(Q_{C\{C|C \in N_i\}, m_i} = Q_{m_i}); \text{ and,}$$

$$A \not\vdash B$$

$$y = f_0(x_1, x_2)$$

$$y_{\{y|y \leq a\}} = f_1(x) \quad y_{\{y|y \geq a\}} = f_2(x_2)$$

FIGURE AII-15

As seen for Figure AII-15, Case 8 is similar to Case 7 in that we always obtain an extraneous or invalid function. Case 8 is invalid for HOS and, thus, cannot be considered a primitive control structure.

Case 9:

$$X_{A\{A|A \in N_i\}, m_i} = X_{m_i} ;$$

$$Q_{A\{A|A \in N_i\}, m_i} \not\subseteq Q_{m_i} ;$$

$$(Y_{B\{B|B \in N_i\}, m_i} \not\subseteq Y_{m_i}) \rightarrow (Y_{C\{C|C \in N_i\}, m_i} = Y_{m_i}) \rightarrow$$

$$(P_{C\{C|C \in N_i\}, m_i} = P_{m_i}) ; \text{ and,}$$

$$A \neq B$$

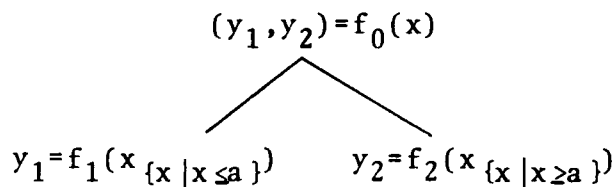


FIGURE AII-16

Consider Figure AII-16: Here, we cannot get a complete output element for any element of x . Case 9 is invalid for HOS, and thus, cannot be considered a primitive control structure

To summarize thus far, the primitive control structures in Table AII-1 have been established.

COMPOSITION	<p><u>CASE 1:</u></p> $X_{A\{A A \in N_i\}, m_i} = X_{m_i}; \quad Q_{A\{A A \in N_i\}, m_i} = Q_{m_i};$ $Y_{B\{B B \in N_i\}, m_i} = Y_{m_i}; \quad P_{B\{B B \in N_i\}, m_i} = P_{m_i}; \quad \text{and } A \neq B$
CLASS PARTITION	<p><u>CASE 2:</u></p> $(X_{A\{A A \in N_i\}, m_i} \neq X_{m_i}) \rightarrow (X_{C\{C C \subset N_i\}, m_i} = X_{m_i}) \rightarrow (Q_{C\{C C \subset N_i\}, m_i} = Q_{m_i});$ $(Y_{B\{B B \in N_i\}, m_i} \neq Y_{m_i}) \rightarrow (Y_{C\{C C \subset N_i\}, m_i} = Y_{m_i}) \rightarrow (P_{C\{C C \subset N_i\}, m_i} = P_{m_i});$ <p>and, $A \neq B$</p>
SET PARTITION	<p><u>CASE 3:</u></p> $X_{A\{A A \in N_i\}, m_i} = X_{m_i}; \quad Q_{A\{A A \in N_i\}, m_i} \neq Q_{m_i}; \quad Y_{B\{B B \in N_i\}, m_i} = Y_{m_i};$ $P_{B\{B B \in N_i\}, m_i} = P_{m_i}; \quad \text{and, } A \neq B$ <p><u>CASE 4:</u></p> $X_{A\{A A \in N_i\}, m_i} = X_{m_i}; \quad Q_{A\{A A \in N_i\}, m_i} \neq Q_{m_i}; \quad Y_{B\{B B \in N_i\}, m_i} = Y_{m_i};$ $P_{B\{B B \in N_i\}, m_i} \neq P_{m_i}; \quad \text{and, } A \neq B$

TABLE 1: THE PRIMITIVE CONTROL STRUCTURE FOR HOS

The following combinations of primitive control structures can be used as abstract control structures for building a system specification. Note that we could not show set partition and class partition on one level (Figure AII-17) because to do so implied the need for local variables. Since local variables imply composition, there is no way to obtain set partition and class partition on one level without composition. In each of the cases shown, the combinations can be regrouped into the original three primitive control structures.

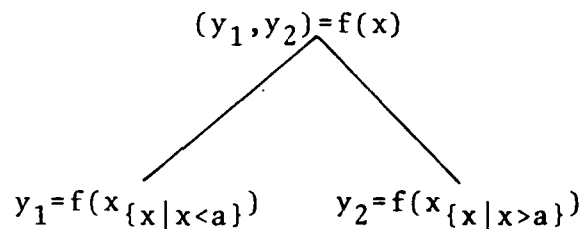


FIGURE AII-17: An Invalid Combination of Set Partition and Class Partition

Examples of Valid Abstract Control Structures Derived from Composition and Set Partition Primitives.

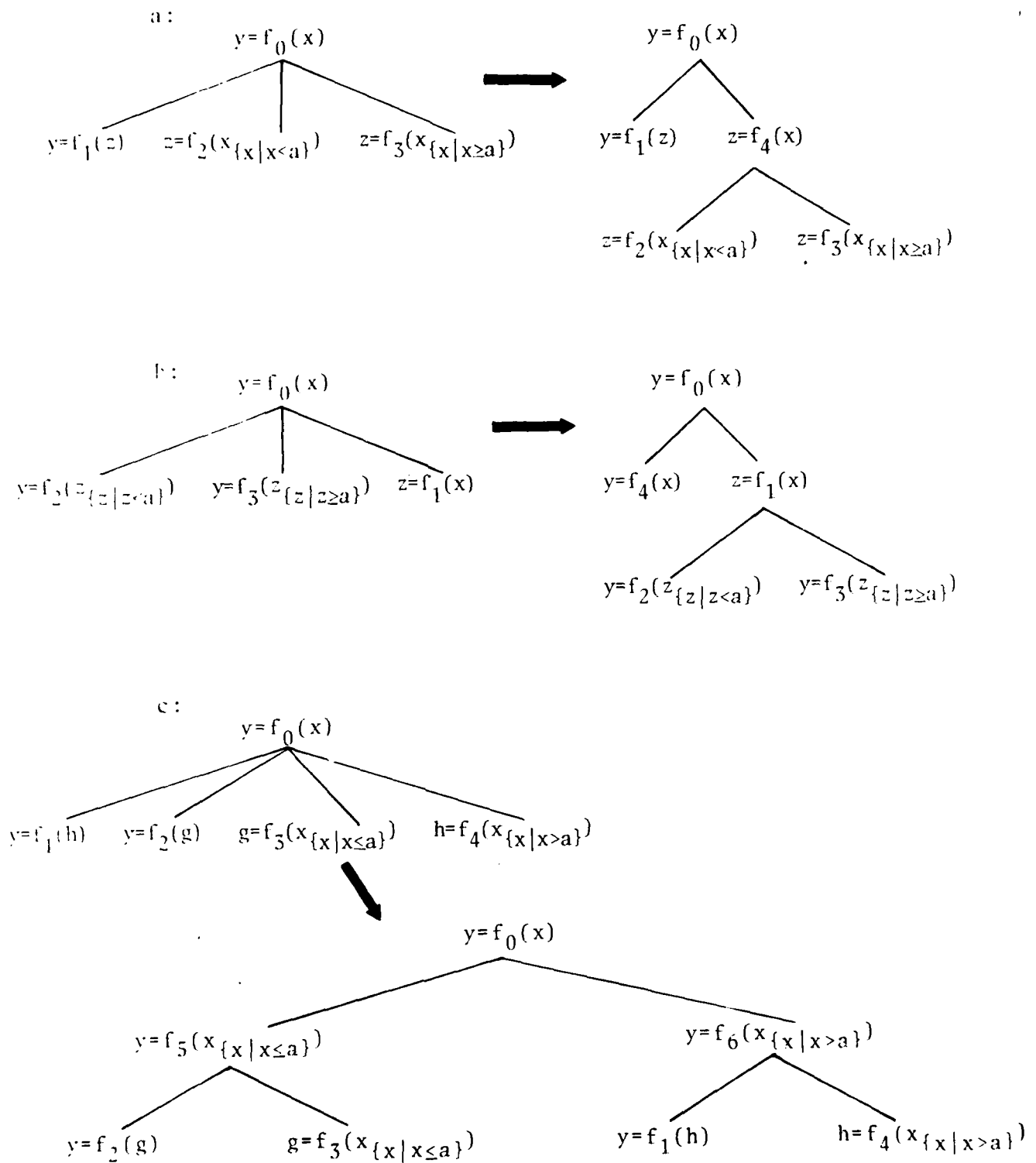


FIGURE AII-18

AII-21

Examples of Valid Abstract Control Structures Derived from Composition
and Class Partition Primitives

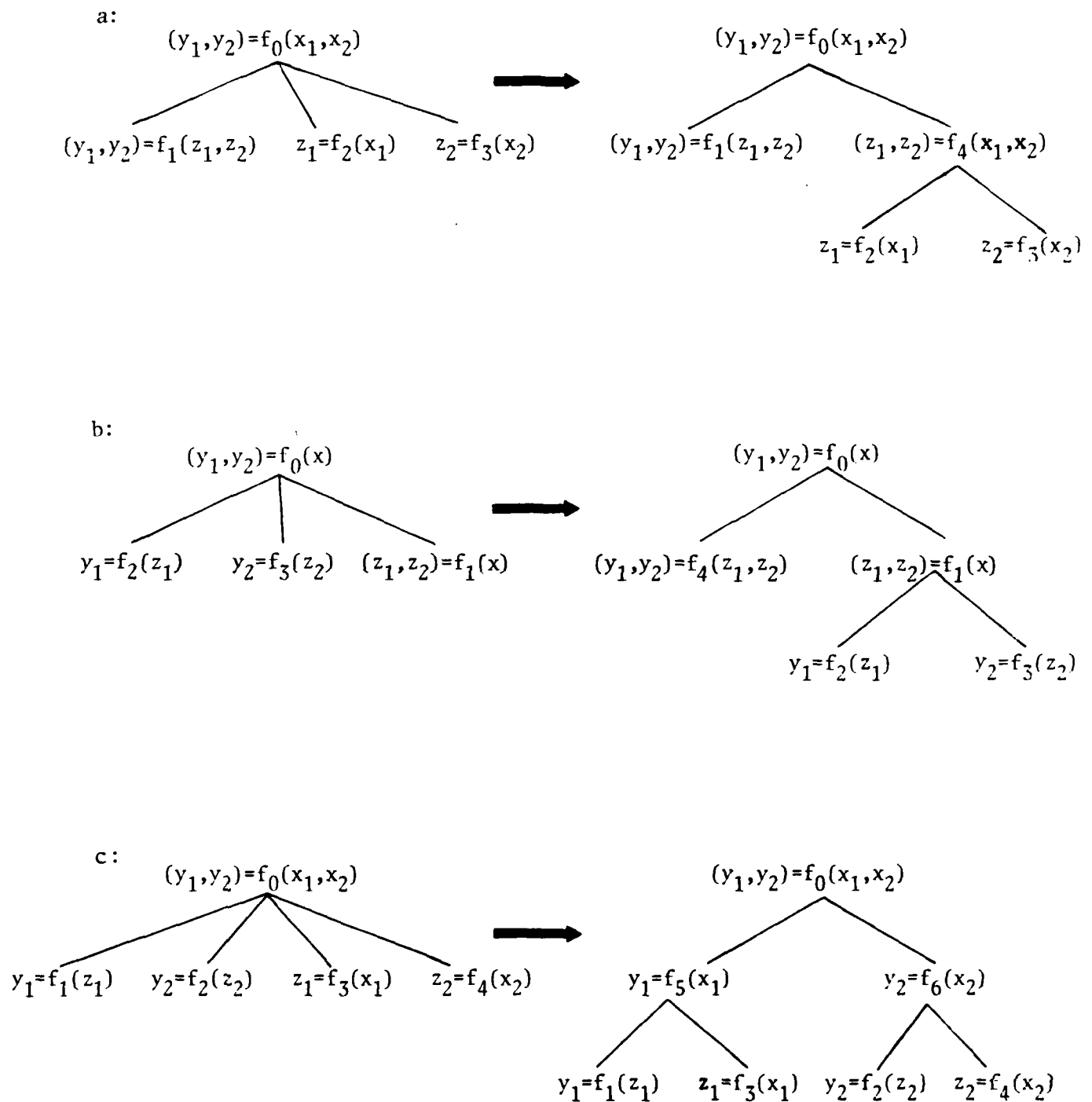


FIGURE AII-19

AII-22